DOCUMENT RESUME

ED 270 092 IR 012 110

AUTHOR Moulton, Peter
TITLE Structured Programming: An Introduction.
INSTITUTION International Council for Computers in Education, Eugene, Oreg.
REPORT NO ISBN-0-924667-27-3
PUB DATE 86
NOTE 85p.
AVAILABLE FROM International Council for Computers in Education Publications, 1787 Agate St., Eugene, OR 97403 ($6.00).
PUB TYPE Guides - Classroom Use - Guides (For Teachers) (052) -- Computer Programs (101)

EDRS PRICE MF01/PC04 Plus Postage.
DESCRIPTORS Computer Science Education; Computer Software; *Microcomputers; Programers; *Programing; *Programing Languages; Secondary Education
IDENTIFIERS *BASIC Programing Language; Debugging (Computers); PASCAL Programing Language; *Structured Programing

ABSTRACT

Designed for use by computer programming teachers, this booklet presents the concepts of structured programming and provides examples of how to implement this methodology, which provides a systematic way of organizing programs so that even large and complex programs are easier to understand and modify than unstructured programs. After a brief introduction which discusses the history of programming languages, this guide presents a program written in an unstructured manner and then proceeds to develop the structured concepts and show how these improve the organization of the unstructured examples in both BASIC and Pascal. It then describes how to use structured programming with languages such as BASIC, which were developed before the structured methodology. After presenting additional examples (including the development of a gradebook program) and some extension to the structure concepts, the guide concludes with an appendix which discusses strategies for informally proving that programs work. (JB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED270092

U.S. DEPARTMENT OF EDUCATION
OERI
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
This document has been reproduced as received from the person or organization originating it.
Minor changes have been made to improve reproduction quality.
Points of view or opinions stated in this document do not necessarily represent official position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY
David Morsund
TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

Peter Moulton has taught computer science for 15 years at the University of Edinburgh, Scotland, University of Oregon and San Jose State University. His primary area of interest has been programming languages and methodology. He has written a textbook and several articles on the subject.

In addition to his doctorate in computer science from the University of Wisconsin, he has a Ph.D. in counseling psychology from the University of Oregon. He is currently working as a psychologist in private practice, and as a consultant in both of his specialty areas.

This booklet is published by the International Council for Computers in Education, a non-profit, tax-exempt professional organization. ICCE is dedicated to improving educational uses of computers and to helping both students and teachers become more computer literate. ICCE publishes *The Computing Teacher*, a journal for teachers and for teacher educators. It also publishes over 15 booklets of interest to educators.

The booklet prices given below are for prepaid orders. On other orders a $2.50 processing charge will be added.

QUANTITY	PRICE (U.S. $)
1-4 copies	$6.00 each
5-9 copies	$5.40 each
10-99 copies	$4.80 each
100+ copies	$4.20 each

Please place your orders with:

INTERNATIONAL COUNCIL FOR
COMPUTERS IN EDUCATION
University of Oregon
1787 Agate St.
Eugene, OR 97403-1923
(503) 686-4414, 686-4429

Copyright © ICCE 1986
ISBN 0-924667-27-3

Table of Contents

Preface 3
Chapter 1: Introduction 5
Chapter 2: An Unstructured Program 9
Chapter 3: Top-Down Programming 13
Chapter 4: The Statements of Structured Programming 19
Chapter 5: An Example of Top-Down Programming 29
Chapter 6: Two Additional Examples 37
Chapter 7: Writing Structured Statements in Non-Structured Languages 43
Chapter 8: Translating a Program into BASIC 51
Chapter 9: Extension to Structured Programming Statements 55
Chapter 10: A Project: A Gradebook Program 61
Appendix 69
References 88

Preface

Do you remember when you first learned to program? Did you attend a formal class, or did you pick it up from a friend, or did you buy a programming book and learn it on your own? In any of these cases the chances are very good that the first thing you learned was what a computer can do, for example, perform arithmetic operations, print information on a display or printer, and input values from a keyboard or cards. Along with learning about what a computer can do, you likely learned certain statements of a programming language which would tell the computer to perform certain actions.

And then you started writing programs. When the programs didn't work, you had to "debug" them with the aid of reference manuals, help from friends, and trial and error. In most cases debugging a program would take at least as long as writing the first version. Wouldn't you love to have programs work the first time?

For most people who have learned to program, this process seems very natural, and few might ask for a better way. It does seem natural to acquire the tools and understand their function before we can use them effectively, and so we do need to learn about the capabilities of computers. But after this first step, there is a better way.

Writing a program is not just a matter of telling the computer to do something; it entails a significant problem-solving process. We must analyze exactly what we want the computer to do and then organize the instructions to the computer in such a way that the computer will do what we want it to do. Errors in a program do not occur because of failures of the computer. They result when we tell the computer to do something which is not what we want it to do. Most often they occur when we do not give sufficient attention to analyzing the problem we want the computer to solve for us.

The traditional approach to learning to program does not include a systematic method of analyzing our problems before we create a set of instructions for the computer. In some situations flowcharting is taught, but the traditional style of flowcharting simply substitutes boxes for instructions and produces a graphic representation of the program. It does not help us organize our own thinking processes. As programs become large and

complex, the major problem in programming is coping with the limitations of our own thinking processes.

Structured programming is a programming methodology which helps us solve this problem of limitations. It has been developed over the past 15 years and during this period it has significantly changed the way in which programs are written and has influenced the development of several new programming languages such as Pascal. The increasing use of Pascal in schools is evidence of the advantages and power of structured programming.

This booklet will present the concepts of structured programming. It will begin with a program written in an unstructured manner and then proceed to develop the structured concepts and show how these improve the organization of the unstructured example. It will then describe how to use structured programming with languages such as BASIC which were developed before the structured methodology. After presenting some more examples and some extensions to the structured concepts, it will present in an appendix strategies for informally proving that programs work.

Hopefully this booklet will change the way you think about writing programs. If so, it will allow you to write programs which run correctly with fewer debugging runs, and which will be easier to understand and modify. One of the proofs of the pudding with this methodology is the number of times students come up with a program and excitedly exclaim, "It worked on the first try!" If you are teaching programming, hopefully this booklet will change the way you teach programming. And if so, it will help students write programs which run with fewer debugging runs and are more easily understood not only by themselves but by you, the teacher.

I would like to acknowledge the helpful suggestions, comments and reactions given by a group of teachers in the Anchorage, Alaska School District who served as a first audience for most of this material. I would also like to give appreciative recognition to Professor David Moursund for his valuable suggestions, ongoing encouragement, and immeasurable patience.

1

Introduction

In less than a lifetime, computers have been born and have grown into a major influence in our lives. They have changed our ways of thinking, and while making our lives more comfortable, they are challenging us with increasingly complex problems to solve. The complexity of the programs we are able to create and have computers run stretches the limits of our thinking.

Computers as we know them date from World War II. The first computer was constructed with electromechanical relays — actual physically moving switches — to perform the storage of data and the switching of the circuitry. Mechanical devices are slow and require extensive maintenance. Even so, this first computer was able to outperform a roomful of people working with desk calculators. A second computer built during the war made use of vacuum tubes to perform the switching and store data. The commercial computers built during the 1950s continued to use vacuum tubes. These computers were physically large and required ongoing maintenance. During the early 1960s the transistor replaced the vacuum tube. The first transistors were discrete components; that is, they were separate entities which had to be placed on a printed circuit board and soldered into position. Thus, computers were essentially handbuilt and were very expensive. During the late 1960s and the 1970s the development of integrated circuit technology allowed entire circuits to be mass produced on silicon wafers using a photoengraving process. Now most of the circuitry in a computer is produced on these chips of silicon, and the cost of computing power has decreased dramatically. A level of computing power that would have cost $50,000 fifteen years ago can now be purchased for $1,000. And the maintenance has become almost negligible.

While the computer hardware revolution has been occurring, another less widely publicized revolution has also been taking place. This revolution has dramatically changed the way in which we write programs.

Every computer is built with a certain set of fundamental operations which perform arithmetic operations, move data into and out of memory, and perform input-output operations. Programs are made up of sequences of these instructions. Twenty-five to thirty years ago programmers had to

learn the numeric codes which would direct the activity of a computer. Everything in a computer is, of course, represented as a number, and in the early days of computers, programmers had to learn codes similar to:

10 n : ADD the number in storage location n to the number in the *accumulator*. (The accumulator is a special storage device in which arithmetic operations can be performed.)

12 n : MOVE the number in storage location n to the accumulator.

13 n : STORE the number in the accumulator into the storage location n.

15 n : SUBTRACT the number in storage location n from the number in the accumulator.

Then in order to have the computer perform the computation indicated by the expression C = A + B - D, a programmer would have to allocate storage locations for values A, B, C, D (say locations numbered 35,36,37,38, respectively) and generate the following instructions:

12 35 (Move A to the accumulator)
10 36 (ADD the value of B to the accumulator)
15 38 (Subtract the value of D from the accumulator)
13 37 (STORE the value of the accumulator into C)

As you can see this was a very detailed and tedious process which required a programmer to be involved with much more than the understanding of the problem being solved.

In order to simplify the programming process and to make it less error prone, mnemonic codes were associated with the numeric codes. Special programs were written to translate the mnemonic codes into the numeric codes. These translating programs were called *assemblers*. They allowed programmers to write their programs in *assembler* language. In assembler language, a programmer might write the sequence of instructions in the above example as:

```
MOVE A
ADD B
SUB D
STORE C
```

The assembler would input these four lines and output the equivalent numeric sequence given above.

While assembler language does make the meaning of each instruction more apparent, it does not help us understand the meaning of the sequence of the instructions. The expression C = A + B - D is much closer to the kinds of expressions which we normally use to describe the problems we are dealing with. These are the kinds of expressions incorporated in *high level* programming languages. The generic name of the programs which translate these high level expressions into the detailed instructions of the computer is *compiler*. Compilers were first developed in the late 1950s and early 1960s. BASIC, FORTRAN, and COBOL are some of the early programming languages.

All of the early programming languages are designed more from the standpoint of the computer than from the standpoint of the programmer. Early on when we didn't know very much about what made a program a good program, it made sense to design a programming language which would essentially present the features of the computer to a programmer and allow the programmer to manipulate them symbolically. Thus all of these languages incorporated the fundamental concepts of performing arithmetic operations, performing input and output, and moving data about within the computer's memory. These languages also incorporated the computer's operations for controlling the sequence of operations and for organizing data. However, computers are designed to solve problems differently from the way people solve problems, and these early programming languages generally required programmers to adapt to the needs of the computer.

As the computer industry matured, it became painfully aware of the high cost of developing software and of the high cost of modifying software. These costs relate not to the difficulty which the computer has in carrying out a program, but the difficulty the programmer has in solving the original problem, the difficulty in organizing the data, and the difficulty in determining the sequence of the operations in a program. This has led to the demand for programming methodologies and programming languages which will produce programs which are easier to understand and easier to modify. This new methodology is referred to as *structured programming*.

An early and leading figure in the development of this new programming methodology is Edsger Dijkstra, a mathematician and computer scientist working in the Netherlands. In the latter half of the 1960s, Dijkstra became concerned with the considerable imbalance between the limited capabilities of the human brain and the enormous complexities of computer programs. While the problems which the programs solve may seem well within the capabilities of human intelligence, the level of detail and the large number of pieces of information which must be dealt with at one time, exceeds the capacity of the human mind. According to a commonly accepted rule in psychology, the human mind is capable of attending to "seven plus or minus two" pieces of information at one time. The program for a compiler may

consist of about 10,000 instructions. The programs for large computer operating systems or for large computer systems such as an airline reservation or air defense system may approach a million instructions.

The problem, as Dijkstra and others saw it, is that the computer quickly leads us into creating programs which are too complex for us to understand. Of course such programs are error prone, difficult to modify, and vulnerable to errors when modified. Dijkstra was concerned with developing methods which would allow us, if not force us, to write "intellectually manageable" programs. His attack on the problem was in two directions. The first was to develop a systematic methodology for organizing programs in order to keep them from getting out of hand. This methodology provides us a way of going about analyzing problems and developing solutions to problems. The second direction was to identify a set of programming constructs which would be understandable and which would be sufficient for programming whatever a computer could compute. A construct would be considered understandable if a strategy could be given for proving that computations specified in terms of that construct indeed computed what we wanted to compute.

Several programming languages have been developed in the last decade which incorporate this new programming methodology. Among these, one of the most popular is Pascal which is becoming available on most of the microcomputers. At the same time BASIC, which is an unstructured language, continues as the most common language on microcomputers. BASIC has developed a very strong foothold and is a simple and easy language to use. While this text will illustrate the advantages of Pascal, it will focus primarily on structured programming with BASIC, for this is one area in great need of better programming methodology.

2

An Unstructured Program

When people first encounter the concept of structured programming, it is often difficult for them to grasp the significance of the methodology. This is often because they have not grappled with programs which are sufficiently large and complex to demand systematic organization. Also, any presentation of structured programming concepts must be of manageable size and therefore cannot present large, complex examples. Most of the software currently written, however, is quite complex and many of the standard kinds of programs such as word processors or simulation programs will contain thousands of instructions. Software systems in industry may contain millions of instructions. Of course it is needless to say that no one person can grasp all of this detail at one time, and systematic programming methods are absolutely necessary if programs of this size are to be developed and maintained.

In order to offer some taste of the value of structured programming, this section will present an example of an unstructured program which you will then be asked to modify. The example will be necessarily small, but hopefully the process of making modifications to this unstructured program and then comparing this process with that of modifying a structured program will be useful in understanding the value of the structured approach.

The following unstructured program is a simple drill and exercise program. It will present a question to a student, accept a response, compare the response to the correct answer, and then give feedback to the student. It is written in BASIC, and the questions and answers are incorporated as DATA statements. It is a simple program and should be easily understood with the aid of the remarks included in the body of the program. Take sufficient time to examine the program to be certain you understand how it works.

```
100 REM GET NEXT ITEM
200 READ Q$
300 REM TEST FOR END
400 IF Q$ = "END" THEN 2000
500 REM GET CORRECT ANSWER
600 READ C$
```

```
700  REM DISPLAY ITEM
800  PRINT Q$
900  REM OBTAIN RESPONSE
1000 INPUT R$
1100 REM COMPARE RESPONSE WITH CORRECT ANSWER
1200 IF R$ = C$ THEN 1700
1300 REM INCORRECT RESPONSE
1400 PRINT "YOUR ANSWER IS NOT CORRECT"
1500 PRINT "THE CORRECT ANSWER IS ";C$
1600 GOTO 100
1700 REM CORRECT RESPONSE
1800 PRINT "YOU ARE RIGHT, VERY GOOD!"
1900 GOTO 100
2000 REM END OF EXERCISE
2100 PRINT "YOU HAVE COMPLETED THE EXERCISE"
2200 STOP
2300 REM ITEMS
2400 DATA "WHAT IS THE CAPITAL OF OREGON","SALEM"
2500 DATA "WHAT IS THE CAPITAL OF NEVADA","CARSON CITY"
2600 DATA "WHAT IS THE CAPITAL OF MONTANA","HELENA"
2700 DATA "WHAT IS THE CAPITAL OF ILLINOIS","SPRINGFIELD"
2800 DATA "END"
2900 END
```

Now that you understand the program, make the following modifications. Do each modification separately and monitor the thinking processes that you use to construct the modification.

Modification 1: The program does not allow the student to make a second attempt to answer the question correctly. If the first response is not correct, the program indicates that it is incorrect and immediately gives the student the correct answer. Modify the program to allow the student a second response if the first one is incorrect. If the second attempt is still incorrect then have the program display the correct answer.

Modification 2: Replace modification 1 with a modification which will allow the student two retries before displaying the correct answer.

Modification 3: Leaving modification 2 intact, modify the program so that it will keep a score for each question answered correctly. Count 1 for each correct response regardless of whether the correct response was the first or second or third attempt for a given question. At the end of the exercise display the number of correct responses and the number of questions. For example if 15 questions were presented and the student was unable to answer 3 of them, output the following:

```
12 QUESTIONS OUT OF 15 WERE ANSWERED CORRECTLY
```

Modification 4: Replace modification 3 with a modification which will increment the score by 3 if it is answered correctly on the first attempt, by 2 if answered correctly on the second attempt, and by 1 if it is answered correctly on the third attempt (the second retry). At the end of the program, output a message giving the student the score obtained. For example if 15 items were presented (giving a maximum score of 45) and the student obtained a score of 30, output:

```
YOUR SCORE IS 30 OUT OF A POSSIBLE 45 POINTS
```

Modification 5: Leaving modification 4 intact, modify the program so that it will obtain the student's name and the date, and at the end of the exercise will print a teacher's report containing the date, the student's name, and the score obtained.

3

Top-Down Programming

The method developed by Dijkstra and others for the organization of programs is commonly called *top-down programming* or *stepwise refinement*. Another term commonly used is *structured programming*. While structured programming implies the use of a top-down methodology, it usually implies the use of a programming language which incorporates the statements of the kind presented in Chapter 4. Top-down programming begins a program with a single statement of what we want to do. In a very general way we might begin all of our programs as the single statement

```
DO IT
```

If this statement indeed expresses what we want the computer to do, then we know that the performance of this statement will in fact produce the desired computation. The obvious rub comes, of course, from the fact that there are no computers which can accept the statement "DO IT" and perform the particular computation we have in mind. If what we want to do is to print the product of 1234 and 5678, then many computers will accept the statement

```
PRINT 1234 * 5678
```

and will perform the intended computation. Whenever the computer cannot accept a statement, we must decompose the statement into a set of simpler statements. However, and this is a most important point, *we must decompose a statement in such a way that we know that the decomposition will lead to the same computation which was intended by the original statement*. We must be careful to decompose a statement into a set of statements no larger than we can intellectually manage. This implies that the statements of the decompostion may not yet be at a level which can be accepted by the computer and will have to be decomposed further. This process of successively refining each statement into a set of more detailed

statements is called *stepwise refinement*. In outline this process goes as follows:

```
DO IT
    \ STEP 1
          \ STEP 1.1
            STEP 1.2
            STEP 1.3
    \ STEP 2
          \ STEP 2.1
            STEP 2.2
```

Of course there may be any number of steps at each level and any number of levels until the program is completely developed. What is important is that at each level we know that the computation intended by DO IT is the computation invoked by STEP 1 followed by STEP 2 and that the computation intended by STEP 1 is the computation invoked by STEP 1.1 followed by STEP 1.2 followed by STEP 1.3 and so on.

As a specific example consider the problem of writing a program to compute the cost of shipping containers of grain. The first statement of the problem might be

```
COMPUTE SHIPPING COST
```

Most computers will not be able to interpret and execute this statement, and we will have to refine it. Assume that the cost of shipping a container is computed as a rate per ton for each hundred miles shipped. Then we will have to compute the weight of the container, obtain the distance shipped, obtain the rate per ton, and then finally compute the cost. This refinement of the problem can then be written as:

```
COMPUTE SHIPPING COST
  1. COMPUTE WEIGHT OF CONTAINER
  2. OBTAIN DISTANCE SHIPPED
  3. OBTAIN RATE
  4. COMPUTE COST
```

Of course this refinement represents the author's understanding of the problem. Based on a different understanding of the problem, others might arrive at a different sequence.

Since most programming languages which can be interpreted by the computer do not include statements such as these, we will have to continue with the decomposition or refinement process. The weight of the container will be determined by the volume of the container and the weight of the

grain per unit of volume. The first statement in the above sequence can then be refined into:

1. COMPUTE WEIGHT OF CONTAINER
   1.1 COMPUTE VOLUME OF CONTAINER
   1.2 OBTAIN WEIGHT PER UNIT VOLUME
   1.3 COMPUTE WEIGHT

The example is sufficiently simple that we should be able to argue informally that the refinement will in fact compute what we intend by the statement COMPUTE WEIGHT OF CONTAINER. Now, continuing another step, we can refine STEP 1.1 into the following two statements:

1.1 COMPUTE VOLUME OF CONTAINER
   1.1.1 OBTAIN DIMENSIONS
   1.1.2 COMPUTE VOLUME

An important advantage of this method of developing a program is that it allows us to focus upon one piece of the program at a time without losing track of where we are in the overall program. We know that all of the statements which are produced as a refinement of a given statement will occur in the program after all of the statements refined from the statement occurring before the given statement and before all of the statements refined from the statement occurring after the given statement. Thus in the above example all of the statements refined from Statement 2, OBTAIN DISTANCE SHIPPED will occur after all of the statements refined from Statement 1, COMPUTE WEIGHT OF CONTAINER (in particular after 1.3 COMPUTE WEIGHT) and before any statements refined from Statement 3, OBTAIN RATE . We can observe this by putting all of the above statements together as:

COMPUTE SHIPPING COST
   1. COMPUTE WEIGHT OF CONTAINER
      1.1 COMPUTE VOLUME OF CONTAINER
         1.1.1 OBTAIN DIMENSIONS
         1.1.2 COMPUTE VOLUME
      1.2 OBTAIN WEIGHT PER UNIT VOLUME
      1.3 COMPUTE WEIGHT
   2. OBTAIN DISTANCE SHIPPED
   3. OBTAIN RATE
   4. COMPUTE COST

In using this numbering scheme with large programs, the number of a particular statement may become unwieldy. An abbreviation convention which will be used in this text is to write only the right-most digit preceded

by a period. A statement number preceded by a period informs the reader that in order to obtain the complete number, the lines occurring before the given line must be scanned in order to accumulate the complete number. With very large programs, the complete number of the statement occurring at the top of the page or the beginning of each major section may be written in full. Using this abbreviation scheme, the previous program would become:

```
COMPUTE SHIPPING COST
  1. COMPUTE WEIGHT OF CONTAINER
    .1 COMPUTE VOLUME OF CONTAINER
      .1 OBTAIN DIMENSIONS
      .2 COMPUTE VOLUME
    .2 OBTAIN WEIGHT PER UNIT VOLUME
    .3 COMPUTE WEIGHT
  2. OBTAIN DISTANCE SHIPPED
  3. OBTAIN RATE
  4. COMPUTE COST
```

While this approach to creating programs may seem obvious and of no great significance when considering small programs written by a single programmer, as programs become complex and are developed by teams of programmers, this organizational discipline becomes considerably more significant. The "outline" numbering scheme helps keep track of the levels and pieces of the refinements. Also notice how the indentations help us understand the refinements.

The next refinement step in this example can be to the level of statements in an actual programming language which the computer can understand. Using the language BASIC we can obtain the following program. It should be noted that this program has been kept very simple in order to focus on the structure. In practice more helpful interaction with the user would be desirable.

```
100  REM COMPUTE SHIPPING COST
200  REM       1.COMPUTE WEIGHT OF CONTAINER
300  REM           1.1 COMPUTE VOLUME OF CONTAINER
400  REM               1.1.1 OBTAIN DIMENSIONS
500                          PRINT "INPUT LENGTH,WIDTH,HEIGHT"
600                          INPUT L,W,H
700  REM               1.1.2 COMPUTE VOLUME
800                          V = L*W*H
900  REM           1.2 OBTAIN WEIGHT PER UNIT VOLUME
1000                         PRINT "INPUT WEIGHT PER UNIT VOLUME"
1100                         INPUT P
1200 REM           1.3 COMPUTE WEIGHT
1300                         W = V*P
```

```
1400 REM    2.OBTAIN DISTANCE SHIPPED
1500                        PRINT "INPUT DISTANCE"
1600                        INPUT D
1700 REM    3.OBTAIN RATE
1800                        PRINT "INPUT RATE"
1900                        INPUT R
2000 REM    4.COMPUTE COST
2100                        C = W*D*R
2200                        PRINT "COST = ";C
2300 END
```

Notice how the statements at the higher levels have been included as remarks in the program and how the BASIC statements have been offset to allow the higher levels to be more visible. This has been done because our understanding of the intention of the program will come more easily when we read the higher level statements than when we read the detailed BASIC statements. This method of documenting a program allows us to understand the role of any of the detailed levels in the context of the higher levels. The level of higher level statements which should be used for remarks varies with the size and complexity of the program. In this particular example the remarks at lines 400 and 700 might have been left out without loss of understanding. Depending on the reader of the program, the remarks at lines 300, 900 and 1200 might also be omitted.

Exercises

1) Write a program which will compute the change to be given for a purchase. Given the amount of money submitted and the purchase price, compute the number of $50 bills, $20 bills, $10 bills, $5 bills, half-dollars, quarters, dimes, nickels, and pennies to be given in change. Write the program to give the smallest number of coins and bills (e.g. one dime and one nickel rather than three nickels) If you are using a different system of currency, write the program for that currency.

2) The number of characters in a manuscript is approximately the number of pages multiplied by the number of characters per page. The number of characters per page is approximately the number of characters per line multiplied by the number of lines per page. The number of characters which can be stored on a diskette is equal to the number of data tracks per diskette multiplied by the number of characters per track. The number of data tracks per diskette is the number of tracks per diskette less the number of control tracks per diskette. The number of characters per track is equal to the number of sectors per track multiplied by the number of characters per sector. Using a top-down approach, write a program which will compute the number of diskettes required to store a given manuscript.

4

The Statements of Structured Programming

A casual examination of the program of the preceeding section shows that it is a very simple program. There are no loops or branches; the computation begins with the first statement of the program and continues through in a straight line until the last statement has been executed.

Most programs contain GOTOs and IFs. For example, consider the program of Chapter 2. GOTO and IF statements allow us to introduce great complexity into our programs. In fact they allow us to construct unmanageable mazes which frustrate any attempt at defensible understanding. Realizing that existing languages allow us to create such monstrosities, Dijkstra set out to identify a set of programming statements which would be sufficient to describe any computation which the computer could perform and which would also be simple enough to allow us to understand them. By understand them he meant be able to defend programs written using them — in other words, to be able to answer the question "How do you know it works?" at whatever level of rigor might be required. Dijkstra identified three programming statements which are sufficient to describe computations and for which he could specify strategies for proving that the programs written with these statements worked correctly. He named these statements sequence, selection and repetition.

Sequence statements

Sequence statements are simply sequences of statements such as

```
s1;s2;s3;s4
```

or

```
s1
s2
s3
s4
```

where semicolons separate the statements in the first sequence and where individual statements appear on separate lines in the second sequence. In both of these examples a statement in a sequence, such as s1, can be an

assignment statement, an input/output statement, a subroutine call, or one of the sequence, selection or repetition statements. A sequence may be written in flowchart notation as follows:

The following is a sequence in BASIC:

```
300   A=100
400   X=A+B*23
500   READ C
600   PRINT "THIS IS A SEQUENCE"
700   GOSUB 27500
```

A sequence in BASIC may not contain GOTO of IF statements. The program computing the cost of a shipment which was presented as an example of top-down programming is, in its entirety, a sequence statement.

Some readers may question the inclusion of subroutine calls within sequences on the grounds that a subroutine call produces a branch of control out of the sequence. However, if we consider that the subroutine call can be replaced by all of the statements in the subroutine and we consider all of the statements in the subroutine as one large compound statement, then we can justify the use of a subroutine call in a sequence. A GOTO or IF statement, on the other hand, may cause a branch of control out of a sequence never to return.

Selection statements

Selection statements are statements which select between alternative actions according to the truth or falsity of a logical expression. The fundamental form given by Dijkstra is:

```
IF e THEN s1 ELSE s2
```

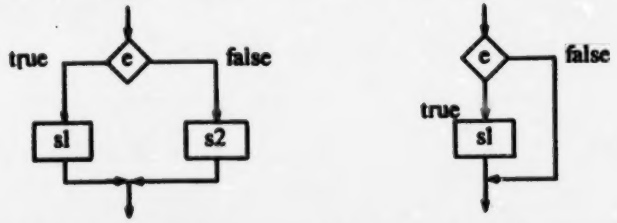
where e is a logical expression and s1, s2 are statements as defined for the sequence statements. If the value of e is true, then the action of the selection statement is s1. Otherwise the value of e must be false and the action of the selection statement is s2. In some cases s2 may do nothing. In this case

20

the statement is written as:

```
IF e THEN sl
```

Selection statements can be written in flowchart notation as follows:



A logical expression is an expression which evaluates to either the value *true* or the value *false*. Some programming languages allow us to assign logical values to variables and then to use these variables as logical expressions. In such a programming language we might write

```
ENDFILE = .TRUE.
```

or `LASTITEM = .FALSE.`

We might then write the selection statement:

```
IF ENDFILE
  THEN PRINT "NO MORE DATA"
  ELSE INPUT A,B,C
```

(Note how the selection statement has been written on three lines with indenting for increased readability.)

Most versions of BASIC do not implement logical variables, and we must assign numeric or string values to variables and then test for these values. For example, we might write:

```
ENDF$ = "TRUE"
```

or `LST$ = "FALSE"`

and then write a selection statement as:

```
IF ENDF$ = "TRUE"
  THEN PRINT "NO MORE DATA"
  ELSE INPUT A,B,C
```


21

Some versions of BASIC do incorporate this form of IF-THEN-ELSE. Since most, however, do not, we will have to translate this form into sequences of more common BASIC statements. This will be done in Chapter 7.

Often, in order to achieve greater efficiency, we would use numbers such as 0 or 1 for true or false rather than string constants. While this may be important in some applications, programmers should be aware that such coding may become obscure and confusing.

The expression ENDF$ = "TRUE" denotes the relationship between the value of the variable ENDF$ and the value of the string constant "TRUE" and evaluates to the value true or false. The equals sign denotes the relation "equal to." The common relations among numeric and string values are:

= *equal to*
<> not equal to
< less than
> greater than
<= less than or equal to
>= greater than or equal to

If X has the value 5 and Y has the value 3, then

X = Y is false
X <> Y is true
X < Y is false
X > Y is true
X <= Y is false
X >= Y is true .

Given logical values of constants, variables or relations, we can apply *logical operators* to compute new logical values. The most common logical operators are: **AND, OR, NOT.** Given that ENDF$="T", X=3, Y=5, Z=-7, then

```
NOT ENDF$="T"      is false
X<Y AND Z<X        is true
Y=X OR X>Z         is true
Y>Z AND Y>X AND X>Z      is true
```

Following are some examples of program fragments using selection statements:

```
a)  IF COLOR$ = "WHITE"
        THEN OPPOSITE$ = "BLACK"
        ELSE OPPOSITE$ = "WHITE"
```

b)
```
LOAD = LOAD +1
    IF LOAD > LAST
        THEN PRINT " THIS IS THE LAST STRAW "
```

c)
```
IF TIME > 12
    THEN TIME = TIME-12
        AMPM$ = "PM"
    ELSE AMPM$ = "AM"
```

d)
```
IF AGE < 12
    THEN IF HEIGHT < 48
            THEN PRINT "LITTLE KID"
            ELSE PRINT "BIG KID"
    ELSE IF AGE < 18
            THEN PRINT "ADOLESCENT"
            ELSE PRINT "ADULT"
```

Exercise

Represent each of (a)-(d) above in flowchart notation.

Repetition Statements

Repetition statements are statements which repeat some action. In order to have the repetition not repeat endlessly, the action is repeated according to the truth or falsity of a logical expression. The fundamental form of the repetition statement given by Dijkstra is:

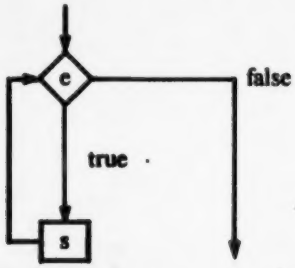
```
WHILE e DO s
```

where e is a logical expression and s is a sequence, selection, or repetition statement. The repetition statement will first test e and then if e is true perform the statement s and then test e and if e is true then perform s and then test e and if e is true then perform s and so on until e is false. When the test of e evaluates to false, the WHILE statement will terminate. It is obvious that if e is true when the first test is made, then the action indicated by s must at some point alter the value of e or else the WHILE will never terminate. The statement, s, is called the body of the repetition.

Another, perhaps more common term for a repetition is *loop*.If a number of statements are to be repeated, then the body of the repetition, s, will be a sequence consisting of those statements. The expression e is called the *termination* condition of the repetition or loop. Except for those circumstances in which one wants to repeat something endlessly, it is always important to know that the termination conditions of loops will be met.

23

A WHILE statement can be written in flowchart notation as:



As an example of a repetition statement, suppose that we wanted to compute shipping costs for a number of containers of grain. The example given in Chapter 3 computes the cost for only one container. We might use subscripts to distinguish individual containers and then change the topmost level statement of that example to

```
COMPUTE SHIPPING COST FOR CONTAINER(I)
```

Then assuming that we have N containers, we would initialize the value of I to 1 and repeat this statement for each of the containers up to N. Using the repetition statement, this would be written as:

```
I = 1
WHILE I <= N DO
  COMPUTE SHIPPING COST FOR CONTAINER(I)
  I = I+1
```

Note how indentation has been used to offset the body of the repetition. When this is done, the word DO becomes unnecessary and in such situations it will often be omitted.

As a second example, suppose that we want to change every occurrence of ";" to "." in a string.

```
SET NEW STRING TO EMPTY
WHILE OLDSTRING NOT EMPTY
  PULL LEFTMOST CHARACTER OFF OF OLDSTRING
  IF CHARACTER = ";"
    THEN CHARACTER = "."
  CONCATENATE CHARACTER TO RIGHT OF NEWSTRING
```

Another common form of the repetition statement is the REPEAT statement which is written as follows:

REPEAT s UNTIL e

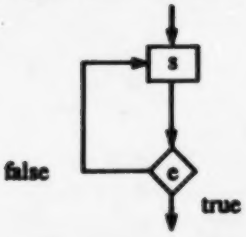
This statement will first perform s and then test e and if e is false perform s and then test e and so on until e is true. A significant difference between the WHILE and the REPEAT statements is that the REPEAT will always perform its action statement at least once. If the termination condition of the WHILE is met upon entry, the action part will not be performed. Of course the sequence

```
s
WHILE e DO s
```

will perform the same computation as

```
REPEAT s UNTIL NOT(e)
```

The flowchart representation of this statement is:



As an example of a repetition statement, suppose that we wanted to perform some error checking on the values input to the program for computing grain shipping costs. In particular suppose that we know that the rate must be between $1.00 and $3.00. Then statement 3. OBTAIN RATE might be refined to:

```
3. OBTAIN RATE
  .1 REPEAT
    .1 REQUEST RATE
    .2 INPUT RATE
    .3 IF RATE<1.00 OR RATE>3.00
            THEN PRINT "RATE ERROR"
  UNTIL RATE>=1.00 AND RATE<=3.00
```

Once again notice how indentation has been used.
BASIC has a FOR-NEXT loop construct which might be confused with the repetition statements just discussed. However, there is an important

25

difference. As long as the index of the FOR-NEXT loop is not altered within the loop and as long as there is no jump (GOTO or IF-THEN) out of the loop, the FOR-NEXT specifies the exact number of times the body of the loop will be executed. This can be viewed as a shorthand method of writing the body of the loop the specified number of times. For example

```
FOR I = 1 TO 4
  SUM = SUM + A[I]
NEXT I
```

can be written as

```
SUM = SUM + A[1]
SUM = SUM + A[2]
SUM = SUM + A[3]
SUM = SUM + A[4]
```

Because of this the FOR-NEXT under the above conditions will be considered an abbreviation of a sequence. FOR-NEXT loops which alter the index within the loop or which jump out of the loop will generally not be considered to be well-structured.

It is useful to note that the flowcharts for each of the structured statements can be placed in a box with one entrance and one exit. This feature results from the disciplined nature of the structured statements. It is an important characteristic when considering ways of proving that programs work.

Exercises

Write structured program fragments for the following:

1) Given arrays, A and B, of 100 elements each, exchange all of the elements of A and B, That is, A(1) and B(1) are exchanged, A(2) and B(2) are exchanged, etc.

2) Given an array, X, of 50 elements
   a) find the largest value in X.
   b) find the largest value in X and the subscript of the first element containing this value.

3) Given a string, STR$, check that every occurrence of "A" is followed by a consonant and that every occurrence of "AN" is followed by a vowel. If an error occurs, output the "A" or the "AN" together with the word which follows (that is, the string of characters up to the next blank).

4) Certain animals are to be graded by color and size according to the following table:

COLOR	SIZE	GRADE
BLACK	LARGE	A
BLACK	SMALL	B
BROWN	BROWN	C
BROWN	BROWN	D

Write a program fragment which will print A,B,C,or D according to the values of variables COLOR and SIZE.

5

An Example of Top-Down Programming

Now that we have all of the structured programming statements and can write structured programs with selection and repetition, let's consider some more examples. To begin, we can consider the drill-and-practice program presented in Chapter 2 and then work through the modifications suggested. If you made these modifications and noted your thought processes while making them, you will be in a good position to compare the process of modifying an unstructured program with the process of modifying a structured program.

To begin with a single statement of the drill-and-practice problem, we might write:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
```

This statement might then be refined into the following two steps:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
  1. INITIALIZE
  2. ADMINISTER EXERCISE
```

The first step of most programs is to initialize the data of the program. In this simple example the initialization will be to input the first question. Thus we could write:

```
1. INITIALIZE
      .1 GET FIRST ITEM
```

This decomposition represents my understanding of the problem. Others with a different understanding might write the program differently. Continuing with this process, the second step might be refined as:

```
2. ADMINISTER EXERCISE
      .1 WHILE ITEM <> "END"
        .1 GET ANSWER TO ITEM
```

```
.2 PRESENT ITEM
.3 OBTAIN AND CHECK RESPONSE
.4 GET NEXT ITEM
```

Again it should be emphasized that this represents a decomposition based upon my understanding of the problem. The third step of the last sequence needs to be refined and might be rewritten as:

```
2.1.3 OBTAIN AND CHECK RESPONSE
        .1 INPUT RESPONSE
        .2 IF RESPONSE = ANSWER TO ITEM
          .1 THEN DISPLAY CORRECT MESSAGE
          .2 ELSE
            .1 DISPLAY INCORRECT MESSAGE
            .2 DISPLAY ANSWER TO ITEM
```

The complete program now appears as:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
  1. INITIALIZE
    .1 GET FIRST ITEM
  2. ADMINISTER EXERCISE
    .1 WHILE ITEM <> "END"
      .1 GET ANSWER TO ITEM
      .2 PRESENT ITEM
      .3 OBTAIN AND CHECK RESPONSE
        .1 INPUT RESPONSE
        .2 IF RESPONSE = ANSWER TO ITEM
          .1 THEN DISPLAY CORRECT MESSAGE
          .2 ELSE
            .1 DISPLAY INCORRECT MESSAGE
            .2 DISPLAY ANSWER TO ITEM
      .4 GET NEXT ITEM
```

Notice the numbering scheme which is being used with selection statements. The "then-part" is numbered .1 and the "else-part" is numbered .2.

Now we can begin making the modifications. The first modification is to allow the student a second attempt before displaying the correct answer to an item. By examining the above program, we can see that the step which must be modified is the ELSE at step 2.1.3.2.2. This is the step which will be executed when the student gives an incorrect response. We can modify this as:

```
2.1.3.2.2 ELSE
          .1 REQUEST RETRY
          .2 OBTAIN AND CHECK RETRY
```

The second step in this last sequence can be refined to:

```
.2 OBTAIN AND CHECK RETRY
  .1 INPUT RESPONSE
  .2 IF RESPONSE = ANSWER TO ITEM
    .1 THEN DISPLAY CORRECT MESSAGE
    .2 ELSE
      .1 DISPLAY INCORRECT MESSAGE
      .2 DISPLAY ANSWER TO ITEM
```

Now the modified program appears as:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
  1. INITIALIZE
    .1 GET FIRST ITEM
  2. ADMINISTER EXERCISE
    .1 WHILE ITEM <> "END"
      .1 GET ANSWER TO ITEM
      .2 PRESENT ITEM
      .3 OBTAIN AND CHECK RESPONSE
        .1 INPUT RESPONSE
        .2 IF RESPONSE = ANSWER TO ITEM
          .1 THEN DISPLAY CORRECT MESSAGE
          .2 ELSE
            .1 REQUEST RETRY
            .2 OBTAIN AND CHECK RETRY
              .1 INPUT RESPONSE
              .2 IF RESPONSE = ANSWER TO ITEM
                .1 THEN DISPLAY CORRECT MESSAGE
                .2 ELSE
                  .1 DISPLAY INCORRECT MESSAGE
                  .2 DISPLAY ANSWER TO ITEM
      .4 GET NEXT ITEM
```

Notice how the structuring helps us locate the section of the program to modify and how it helps us localize or isolate the changes so that the rest of the program is not altered.

Now we can consider the next modification which will allow the student two retries. We could change the ELSE at STEP 2.1.3.2.2.2.2.2. in the same way that we made the last modification. This, however, would require that we add another eight steps to the program. At this point it would be better

to put the retries into a repetition which will repeat until the correct response is given or until the student has made two retries or a total of three responses. In order to do this we will have to keep a count of the number of responses given to the current item. This count will have to be initialized to one when the first response is obtained. This will require adding a step after 2.1.3.1.

```
2.1.3.1 INPUT RESPONSE
      .2 NUMBER OF RESPONSES = 1
      .3 IF RESPONSE = ANSWER TO ITEM
```

Then step 2.1.3 will have to be changed. One way of doing this is to write a repetition which will loop until the correct response is given or until three incorrect responses have been given. After this loop the program could then display the appropriate feedback to the student. This would result in changing 2.1.3 to be the repetition and then adding a new step to display feedback. Adding a new step will require changing the numbering of STEP 2.1.4. These changes could be written as:

```
2.1.3 OBTAIN RESPONSE AND CHECK FOR RETRIES
    .4 CHECK FINAL RESPONSE AND DISPLAY FEEDBACK
    .5 GET NEXT ITEM
```

The first two steps of this last sequence need to be refined. This could be done as follows:

```
2.1.3 OBTAIN RESPONSE AND CHECK FOR RETRIES
      .1 INPUT RESPONSE
      .2 NUMBER OF RESPONSE = 1
      .3 WHILE RESPONSE <> ANSWER TO ITEM
                AND NUMBER OF RESPONSES < 3
        .1 REQUEST RETRY
        .2 INPUT RESPONSE
        .3 INCREMENT NUMBER OF RESPONSES
      .4 CHECK FINAL RESPONSE AND DISPLAY FEEDBACK
        .1 IF RESPONSE = ANSWER TO ITEM
          .1 THEN DISPLAY CORRECT MESSAGE
          .2 ELSE
            .1 DISPLAY INCORRECT MESSAGE
            .2 DISPLAY ANSWER TO ITEM
```

Now the modified program appears as:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
  1. INITIALIZE
    .1 GET FIRST ITEM
```

```
2. ADMINISTER EXERCISE
  .1 WHILE ITEM <> "END"
    .1 GET ANSWER TO ITEM
    .2 PRESENT ITEM
    .3 OBTAIN RESPONSE AND CHECK FOR RETRIES
      .1 INPUT RESPONSE
      .2 NUMBER OF RESPONSE = 1
      .3 WHILE RESPONSE <> ANSWER TO ITEM
                AND NUMBER OF RESPONSES < 3
        .1 REQUEST RETRY
        .2 INPUT RESPONSE
        .3 INCREMENT NUMBER OF RESPONSES
    .4 CHECK FINAL RESPONSE AND DISPLAY FEEDBACK
      .1 IF RESPONSE = ANSWER TO ITEM
        .1 THEN DISPLAY CORRECT MESSAGE
        .2 ELSE
          .1 DISPLAY INCORRECT MESSAGE
          .2 DISPLAY ANSWER TO ITEM
    .5 GET NEXT ITEM
```

Now let's try the modification to accumulate a score of the number of items which a student answers correctly. At first this score will be a count of one for each correct response regardless of the number of retries.

BEFORE PROCEEDING FURTHER YOU MAY WISH TO MAKE THIS MODIFICATION YOURSELF FOLLOWING THE APPROACH WE HAVE BEEN USING.

This modification will accumulate a score over all of the items presented. Therefore it will be initialized before entering the repetition of Step 2. and will be displayed following the completion of Step 2. This means that we will initialize the score as part of Step 1 and that we will display the score in a new Step 3 following Step 2. Within the repetition of Step 2, we will increment the score if the student gives a correct response. We know that a student has given a correct response at Step 2.1.4.1.1. The modification of Step 1 could be as follows:

```
1. INITIALIZE
  .1 GET FIRST ITEM
  .2 SCORE = 0
```

The modification of 2.1.4.1.1 could be:

```
2.1.4.1.1 THEN
          .1 DISPLAY CORRECT MESSAGE
          .2 INCREMENT SCORE
```

The new step following the repetition could be:

```
3. DISPLAY SCORE
```

Since these modifications are brief, the complete program will not be rewritten. However, before continuing, be certain that you can locate the placement of the modifications.

The fourth modification suggested in Chapter 2 was that of computing a weighted score for the student. This score was 3 for a correct response on the first attempt, 2 for a correct response on the second attempt, and 1 for the third attempt. This modification can be implemented by changing the step 2.1.4.1.1.2 and adding the value (4-NUMBER OF RESPONSES) to the score. In addition, in order to include the maximum possible score in the final message to the student, it will be necessary to count the number of items and multiply by three. This will require a new variable, NUMITEM, which will be initialized to zero and incremented as each new item is displayed. This will result in a modified initialization as follows:

```
1. INITIALIZE
  .1 GET FIRST ITEM
  .2 SCORE = 0
  .3 NUMITEM =0
```

The step to increment NUMITEM will result in the following change to Statement 2.1.2 :

```
2.1.2 INITIALIZE AND PRESENT ITEM
      .1 DISPLAY ITEM
      .2 NUMITEM = NUMITEM + 1
```

The last modification is to incorporate a teacher's report which will contain the student's name, the date and the student's score. This will require a modification to the initialization of Step 1 to obtain the student's name and the date. Then the teacher's report can be output either as Step 4 or it could be output as a second step within a revised Step 3. Doing the latter, these modifications would appear as:

```
1. INITIALIZE
  .1 GET FIRST ITEM
  .2 SCORE = 0
  .3 NUMITEM = 0
  .4 OBTAIN STUDENT NAME AND DATE
```

```
3. OUTPUT RESULTS
   .1 DISPLAY STUDENT SCORE
   .2 PRINT TEACHER REPORT
```

The complete program with all of the modifications now appears as:

```
ADMINISTER DRILL AND PRACTICE EXERCISE
   1. INITIALIZE
      .1 GET FIRST ITEM
      .2 SCORE = 0
      .3 NUMITEM =0
      .4 OBTAIN STUDENT NAME AND DATE
   2. ADMINISTER EXERCISE
      .1 WHILE ITEM <> "END"
         .1 GET ANSWER TO ITEM
         .2 INITIALIZE AND PRESENT ITEM
            .1 DISPLAY ITEM
            .2 NUMITEM = NUMITEM + 1
         .3 OBTAIN RESPONSE AND CHECK FOR RETRIES
            .1 INPUT RESPONSE
            .2 NUMBER OF RESPONSES = 1
            .3 WHILE RESPONSE <> ANSWER TO ITEM
                  AND NUMBER OF RESPONSES < 3
               .1 REQUEST RETRY
               .2 INPUT RESPONSE
               .3 INCREMENT NUMBER OF RESPONSES
         .4 CHECK FINAL RESPONSE AND DISPLAY FEEDBACK
            .1 IF RESPONSE = ANSWER TO ITEM
               .1 THEN
                  .1 DISPLAY CORRECT MESSAGE
                  .2 ADD (4-NUMBER OF RESPONSES) TO SCORE
               .2 ELSE
                  .1 DISPLAY INCORRECT MESSAGE
                  .2 DISPLAY ANSWER TO ITEM
         .5 GET NEXT ITEM
   3. OUTPUT RESULTS
      .1 DISPLAY STUDENT SCORE
      .2 PRINT TEACHER REPORT
```

Although this example is very small and not very complex, the process of making these modifications may have illustrated the advantages to the structured programming approach. This methodology helps locate and isolate each of the modifications so that confusion is limited and errors are not introduced by inadvertently modifiying other parts of the program. The example should give some taste of what is meant by "keeping the program

intellectually manageable."

The presentation of the program development as a smooth, constant process flowing from top to bottom might mislead the reader into believing that this is the manner in which programs can be developed with the top-down approach. This is not the case. The development of any but the smallest programs proceeds by trying out proposed partial solutions of the problem and then reworking these and other parts of the program which have already been committed to paper. The process works up and down levels, and the program undergoes considerable revision. It is exactly because of this iterative process that the top-down strategy together with the structured program statements have such great value in developing programs. We need these tools to keep track of the organization of the program throughout the shifts among levels. Whenever the process seems tedious and the programmer becomes impatient with reworking the developing program, it is time to recall that changes made at this level are significantly less costly than changes made to programs which are working but incorrect and in which considerable programming and testing effort have been invested.

Exercises

1) Modify the program so that in addition to the question and answer for each item a "hint" will be read from the DATA. This hint is to be presented before the second retry.

2) Modify the program obtained in exercise (1) to record the number of items answered correctly after the hint was displayed. Output this number on the teacher's report.

6

Two Additional Examples

In this section two additional examples of program development using structured programming will be presented. The first will develop a guessing game which makes use of a common searching technique and the second will develop a sorting program.

A Guessing Game

A number of early computer games depended upon the ability of the program to guess a given number within a certain number of tries. The underlying strategy of all of these games is to divide the range in which the number might lie by half with each guess. This is the strategy of binary searching. While the basic algorithm is quite short, it is sufficiently complex to offer a good example for structured programming. As a starting point we might write:

```
GUESS A NUMBER BETWEEN 1 AND 100
```

This can then be refined into

```
1.START WITH THE FULL RANGE AS THE CURRENT RANGE
2.WHILE THE NUMBER NOT FOUND AND CURRENT RANGE NOT EMPTY
  .1 USE MIDPOINT OF CURRENT RANGE AS GUESS
  .2 IF RESPONSE IS YES
    .1 THEN NUMBER FOUND
    .2 ELSE SET APPROPRIATE HALF AS NEW CURRENT RANGE
3.IF NUMBER NOT FOUND
  .1 THEN PRINT "YOU MUST HAVE MADE A MISTAKE"
```

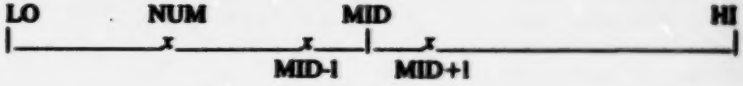
In order to continue, we must decide how we will represent a "range." We can do this by keeping track of the lowest value and of the highest value, and we will know that the possible number will occur somewhere between and including these endpoints. Then the first statement can be refined into

```
1.START WITH THE FULL RANGE AS THE CURRENT RANGE
  .1 LOW = 1
  .2 HI = 100
```

The midpoint of the range is computed as [(LOW+HI)/2] (where [(LOW+HI)/2] is the integer part of (LOW+HI)/2). Statement 2.1 can then be refined into:

```
2.1 USE MIDPOINT OF CURRENT RANGE AS GUESS
    .1 MID = [(LOW+HI)/2]
    .2 OUTPUT GUESS AND OBTAIN RESPONSE
```

In order to select the new current range, we must know whether the number is greater than or less than the guess. If the number is greater than the guess, then the new range will have a lower bound of one greater than the guess. If the number is less than the guess, then the new range will have an upper bound of one less than the guess. This can be seen graphically in the following diagram:



We can then refine 2.2.2 into:

```
2.2.2 ELSE SET APPROPRIATE HALF AS NEW RANGE
      .1 IF NUMBER IS LESS
        .1 THEN HI=MID-1
        .2 ELSE LOW=MID+1
```

The process of selecting half (actually half less one) of the range can continue until either the number is found or until the range is empty in which case we know that the player of the game cheated or forgot the number. We can tell that the range is empty when the value of HI is less than the value of LOW. This situation will arise when the current range consists of one value which will then be equal to MID, and LOW and HI will both be equal to MID. If this is not the number, then either LOW will be set to MID+1 which is greater than HI , or HI will be set to MID-1 which is less than LOW.
We can now put the program together as:

```
GUESS A NUMBER BETWEEN 1 AND 100
  1.START WITH THE FULL RANGE AS THE CURRENT RANGE
    .1 LOW = 1
    .2 HI = 100
  2.WHILE THE NUMBER NOT FOUND AND CURRENT RANGE NOT EMPTY
    .1 USE MIDPOINT OF CURRENT RANGE AS GUESS
      .1 MID = [(LOW+HI)/2]
      .2 OUTPUT GUESS AND OBTAIN RESPONSE
    .2 IF GUESS CORRECT
      .1 THEN NUMBER FOUND
      .2 ELSE
        .1 SET APPROPRIATE HALF AS NEW CURRENT RANGE
        .2 IF NUMBER IS LESS
          .1 THEN HI=MID-1
          .2 ELSE LOW=MID+1
  3.IF NUMBER NOT FOUND
    .1 THEN PRINT "YOU MUST HAVE MADE A MISTAKE"
```

A Sorting Program

Frequently, we are required to arrange the values in a list into either ascending or descending order. A number of algorithms have been devised for performing this task. One of the simplest and most straightforward of these is the bubble sort. As with the guessing game in the previous section, this algorithm is sufficiently complex to illustrate the advantages of structuring, and yet compact enough to provide a manageable example. This program sorts an array by beginning with the whole array and moving or "bubbling" the largest value to the top (ie. to the array position with the largest subscript). This explains its name, "bubble sort." Next the subset of the array consisting of all but the topmost element is considered, and the largest value in this subset is moved to the top of the subset. Then the next subset is all of the current subset except for the topmost element of the current subset, and so on until the next subset consists of a single element. The process can be generalized by considering the whole array as an initial subset. The program becomes:

```
SORT ARRAY
  1. BEGIN WITH THE WHOLE ARRAY AS THE SUBSET
  2. WHILE THE SUBSET IS MORE THAN A SINGLE ELEMENT
    .1 MOVE THE LARGEST VALUE TO THE TOP OF THE SUBSET
    .2 SET NEW SUBSET TO ALL BUT TOPMOST OF CURRENT SUBSET
```

We can refine this program by considering the subscript values of the array. Assume that the first element of the original array is subscripted by the value of FIRST and the last element by the value of LAST. Using LOW and HIGH as the subscripts of the first and last elements of the subset, we can refine Step 1. into the following:

```
1. BEGIN WITH THE WHOLE ARRAY AS THE SUBSET
  .1 LOW = FIRST
  .2 HIGH = LAST
```

The WHILE of Step 2. becomes the following test of subscript values:

```
2. WHILE LOW <> HIGH
```

Step 2.1 can then be refined into:

```
2.1 MOVE LARGEST VALUE TO TOP OF SUBSET
    .1 SET CURRENT TO LOW
    .2 WHILE CURRENT <> HIGH
      .1 IF ARRAY(CURRENT) > ARRAY(CURRENT+1)
        .1 THEN EXCHANGE ARRAY(CURRENT)
             AND ARRAY(CURRENT+1)
      .2 ADVANCE CURRENT TO CURRENT + 1
```

The exchange at Step 2.1.2.1.1 can be refined into:

```
2.1.2.1.1 THEN EXCHANGE ARRAY(CURRENT) AND ARRAY(CURRENT+1)
          .1 TEMP = ARRAY(CURRENT)
          .2 ARRAY(CURRENT) = ARRAY(CURRENT+1)
          .3 ARRAY(CURRENT+1) = TEMP
```

Finally Step 2.2 can be refined into:

```
2.2 SET NEW SUBSET TO ALL BUT TOPMOST OF CURRENT SUBSET
    .1 HIGH = HIGH -1
```

The complete program can now be written as:

```
1. BEGIN WITH THE WHOLE ARRAY AS THE SUBSET
  .1 LOW = FIRST
  .2 HIGH = LAST
2. WHILE LOW <> HIGH
  .1 MOVE LARGEST VALUE TO TOP OF SUBSET
    .1 SET CURRENT TO LOW
    .2 WHILE CURRENT <> HIGH
      .1 IF ARRAY(CURRENT) > ARRAY(CURRENT+1)
        .1 THEN EXCHANGE ARRAY(CURRENT)
                AND ARRAY(CURRENT+1)
          .1 TEMP = ARRAY(CURRENT)
          .2 ARRAY(CURRENT) = ARRAY(CURRENT+1)
          .3 ARRAY(CURRENT+1) = TEMP
      .2 ADVANCE CURRENT TO CURRENT + 1
  .2 SET NEW SUBSET TO ALL BUT TOPMOST OF CURRENT SUBSET
    .1 HIGH = HIGH-1
```

Exercises

1) The basic concept of the guessing game is the same as that used for binary searching . Consider an array, A , of n elements arranged in ascending order. The subscripts will of course range from 1 to n. The values of these elements can be any values with the condition that if i<j , then A(i)<= A(j). Now given a value, V , we want to find the value k such that A(k) = V. This is the same as the guessing game, except rather than asking "is the number k" we will test whether V = A(k). If V is not equal to any of the array elements then we can indicate that it was not found in the same way an error was detected in the guessing game. Rewrite the program of the guessing game to search an ordered array, A, for the index of an element with value, V.

2) The sorting program has an inefficiency which can be easily removed. If the original array is already in order, the program will still cycle through all of the subsets without making any exchanges. Similarly, if the array reaches its final order at some intermediate stage, all of the remaining subsets will be worked through without any exchanges being made. One method of eliminating this unnecessary effort is to record the occurrence of any exchange for a given subset. If no exchanges are made for a given subset, then the sort is complete and the program can terminate. Modify the sorting program to incorporate this change.

7

Writing Structured Statements in Non-Structured Languages

Some of the newer programming languages such as Pascal and some recent versions of BASIC contain control statements which directly implement the structured constructs described in the foregoing material. However, many common languages such as standard BASIC and FORTRAN do not contain these constructs. We must then devise some conventions for translating the structured constructs into these "non-structured" languages. Adopting conventions is important in order to make the translation process as routine and errorfree as possible. Even though these conventions might at times result in apparent inefficiencies, the potential cost of errors is generally higher than the cost of minor inefficiencies. If efficiency is an important issue, a program which is known to work correctly can be carefully modified to make it more efficient. Writing an incorrect but efficient program which is difficult to understand and to correct is of much less value than writing an inefficient but correct program which is understandable and which may be modified more easily.

In order to illustrate some of the advantages of a structured language such as Pascal and in order to show why such languages are gaining increasing acceptance, each of the examples in this section will be written in Pascal as well as BASIC. You should note carefully, however, that a structured language does not guarantee a well-structured program. A structured language incorporates statements corresponding to the structured statements presented in Chapter 4. If these statements are not used in a disciplined manner such as the method of top-down programming, it is quite possible to produce an incomprehensible web of statements. Whether a structured programming language or an unstructured language is used, what is most important is the organization of the program and the way in which the program is developed.

A sequence of statements can of course be implemented as a sequence of statements in the programming language. Almost all programming languages define programs as sequences of statements. At times it becomes valuable to talk about a sequence of statements as a single statement and to name the sequence so that it may be referred to by that name. The most common mechanism for this is the subroutine feature which is available in

most languages. In BASIC we set the sequence of statements apart and refer to the line number of the first line in the sequence with a GOSUB statement. Of course we must then append a RETURN statement at the end of the sequence. For example, in the bubble sort example there is the sequence of statements which exchanges A(CURRENT) and A(CURRENT+1). This sequence could be made available as a subroutine as follows, where I has the value of CURRENT, the subscript of the current element:

```
5000 REM EXCHANGE A(CURRENT) AND A(CURRENT+1)
5010 T = A(I)
5020 A(I) = A(I+1)
5030 A(I+1) = T
5040 RETURN
```

This subroutine could then be referenced as follows:

```
3000 GOSUB 5000
```

In Pascal a sequence of statements is offset with the keywords BEGIN and END. The above sequence can be written in Pascal as:

```
BEGIN (* EXCHANGE A(CURRENT) AND A(CURRENT+1) *)
  T := A[I];
  A[I] := A[I+1];
  A[I+1] := T;
END;
```

Note that ":=" is used rather than "=" for assignment and that each statement is terminated by a semicolon. Also note that comments are enclosed within (* and *) . If this sequence were to be referenced as a subroutine in Pascal, it would have to be declared as a procedure. Each procedure is given a name and is invoked by reference to that name. Thus the sequence could be declared as the procedure EXCHANGE as follows:

```
PROCEDURE EXCHANGE;
  BEGIN (* EXCHANGE A(CURRENT) AND A(CURRENT+1) *)
    T := A[I];
    A[I] := A[I+1];
    A[I+1] := T;
  END;
```

Note that the return is implicit when the end of the sequence is reached. The procedure is invoked by the statement:

```
EXCHANGE;
```

The reader should also be aware that Pascal offers many additional features which will not be discussed in this text. For example Pascal allows parameters to be associated with a procedure reference. In the example just given, good programming practice would have associated the names of the array elements to be exchanged with the procedure reference as:

```
EXCHANGE( A(I) , A(I+1) ) ;
```

However, since this text is primarily concerned with the concepts of structured programming and not with Pascal, only those features of Pascal which illustrate structured programming statements will be described.

The translation of the selection statement must contain a branch around the first alternative if the condition is false and a branch around the second alternative after executing the first alternative when the condition is true. Therefore, translations of selection statements into languages which do not incorporate them must be done with the use of GOTO statements. A standard format for the translation of IF e THEN s1 ELSE s2 is:

```
      IF e THEN L1
      s2
      GOTO L2
L1 s1
L2 _____________________
```

where L1 is the first line number associated with s1 and L2 is the line number immediately following the line or lines associated with s1. In the first version of the drill-and-practice program, the statement

```
2.1.3.2 IF RESPONSE = ANSWER TO ITEM
        .1 THEN DISPLAY CORRECT MESSAGE
        .2 ELSE
          .1 DISPLAY INCORRECT MESSAGE
          .2 DISPLAY ANSWER TO ITEM
```

could be written in BASIC as:

```
6000 IF R$ = A$ THEN 6040
6010 PRINT "INCORRECT ANSWER"
6020 PRINT "THE ANSWER IS ";A$
6030 GOTO 6050
6040 PRINT "CORRECT"
6050
```

Since the alternatives are reversed in sequence this may at times be somewhat confusing, and care must be taken. A good practice is to place

the structured segment as a comment or remark prior to the translation.

Pascal allows the selection statement, and the above example can be written as:

```
IF RESP = ANSW
  THEN WRITELN('CORRECT');
  ELSE BEGIN
    WRITE('INCORRECT ANSWER');
    WRITELN('THE ANSWER IS ',ANSW);
  END
```

If the second or false alternative is null, as in

```
IF e THEN sl
```

the translation pattern becomes:

```
      IF e THEN L1
      GOTO L2
   L1 sl
   L2
```

As an example consider the test for an exchange in the bubble sort program.

```
IF A(I) > A(I+1)
  THEN T = A(I)
    A(I) = A(I+1)
    A(I+1) = T
```

which can be translated as:

```
6000 IF A(I) > A(I+1) THEN 6020
6010 GOTO 6050
6020 T = A(I)
6030 A(I) = A(I+1)
6040 A(I+1) = T
6050
```

The GOTO around the alternative can be eliminated by complementing the value of the test expression. This pattern is:

```
      IF NOT(e) THEN L1
      sl
   L1 _____________________
```

If the programming language does not have the complement operator, NOT, then we must complement the expression ourselves. And we must be careful not to introduce an error in doing so. A common error is to treat A>B as the complement of A<B. Of course NOT(A>B) is the same as A<=B. The above selection statement from the sort program can be translated as:

```
6000 IF A(I) <= A(I+1) THEN 6040
6010 T = A(I)
6020 A(I) = A(I+1)
6030 A(I+1) = T
6040
```

This example can be written in Pascal as:

```
IF A[I] > A[I+1]
  THEN BEGIN
      T := A[I];
      A[I] := A[I+1];
      A[I+1] := T;
    END;
```

The straightforward implementation of the WHILE statement consists of a test which, if true, branches to the first statement of the body of the loop followed by a branch around the body of the loop. The body of the loop must be immediately followed by a branch back to the test. The pattern for WHILE e DO s is:

```
L1 IF e THEN L2
   GOTO L3
L2 s
   GOTO L1
L3 _____
```

where L1 is the line of the test, L2 the first line of the body of the loop, and L3 the first line of the statement following the body of the loop. For example, consider the inner loop of the bubble sort program:

```
2.1.2 WHILE CURRENT <> HIGH
      .1 IF ARRAY(CURRENT) > ARRAY(CURRENT+1)
        .1 THEN EXCHANGE ARRAY(CURRENT) AND
              ARRAY(CURRENT+1)
          .1 TEMP = ARRAY(CURRENT)
          .2 ARRAY(CURRENT) = ARRAY(CURRENT+1)
          .3 ARRAY(CURRENT+1) = TEMP
```

This statement is translated into the following BASIC statements where I is used for the value of CURRENT, H is used for the value of HIGH, and T is used for the value of TEMP, and A is used for ARRAY:

```
7000 IF I <> H THEN 7020
7010 GOTO 7080
7020          IF A(I) <= A(I+1) THEN 7060
7030                   T = A(I)
7040                   A(I) = A(I+1)
7050                   A(I+1) = T
7060          I = I+1
7070 GOTO 7000
7080 _____
```

Just as the GOTO around the statement of a single alternative selection statement was eliminated by complementing the test condition, so the GOTO around the body of the loop can be eliminated by complementing the termination condition. The pattern of this is:

```
L1 IF NOT(c) THEN L2
   s
   GOTO L1
L2 _____
```

and the implementation of the above repetition is:

```
7000 IF I = H THEN 7070
7010          IF A(I) <= A(I+1) THEN 7050
7020                   T = A(I)
7030                   A(I) = A(I+1)
7040                   A(I+1) = T
7050          I = I+1
7060 GOTO 7000
7070 _____
```

While elimination of the GOTO in this way may result in a somewhat shorter, more efficient program, in many cases the transformed logic may be a source of error and the modification may not be intuitively understandable.

Pascal has a WHILE statement and we can write the above example as:

```
WHILE I <> H DO
  BEGIN
    IF A[I] > A[I+1]
      THEN
```

```
        BEGIN
          T := A[I];
          A[I] := A[I+1];
          A[I+1] := T;
        END;
    I := I+1;
END;
```

Note the required keyword, DO, following the termination condition of the WHILE.

The pattern for the REPEAT statement consists of the body of the loop followed by the test which, if true, branches to the next statement. The test is immediately followed by a branch back to the first statement of the body of the loop. Thus, the translation of REPEAT s UNTIL e is:

```
L1 s
   IF e THEN L2
   GOTO L1
L2 _____
```

Once again, the GOTO can be eliminated by complementing the test condition. In this case the test will branch to the beginning of the body of the loop.

```
L1 s
   IF NOT(e) THEN L1
   _____
```

Pascal allows the use of the REPEAT exactly as we have been using it as a structured statement.

Recall that the sequence statements were restricted to sequences of assignment, input/output, subroutine call, selection, and repetition statements. This implies that branch statements such as the GOTO, or IF statements of BASIC are not allowed except as they are employed in the controlled patterns which implement selection and repetition statements. That GOTOs are not generally allowed, and in fact would not be allowed at all if the selection and repetition statements were directly provided in our programming language, has led to the term "goto-less" programming. GOTOs have gotten a bad name in computer science because they allow undisciplined programming styles which are difficult to understand, costly to modify, and which are more error prone than the disciplined style enforced by using only the statements which have just been described. The major problem with GOTOs is that they allow complex webs of branching which frustrate any attempt to keep track of the state of the computation. This of course makes it extremely difficult if not impossible to prove any assertions

regarding the computation. Consider the following pattern of GOTOs and statements:

```
L1 GOTO L8
L2 _____
   _____
L3 GOTO L8
L4 _____
   _____
L5 GOTO L8
L6 _____
   _____
L7 _____
L8 s
   _____
L9 GOTO L8
```

When the statement, s, at L8 is executed, we have no way of knowing whether the statement executed immediately before s is the GOTO at L1, or L3, or L5, or L9, or the statement at L7. Since each of these possibilities will generally have produced different values of variables in the program, in order to consider any assertion made regarding what happens at L8, we would have to consider all of these antecedent possibilities. If each of these antecedent sections of code had eight possible antecedents, we would then have to consider 8x8 or 64 possibilities. In a program of any size, matters would quickly get out of hand. Each of the constructs — sequence, selection, repetition — has only a single entry and a single exit. This enforces a more manageable level of complexity in our programs.

8

Translating a Program into BASIC

We are now in a position to refine programs into the statements of a programming language. As an example, the drill-and-practice program will be refined into Microsoft BASIC. Comments pertaining to particular features of this language will be made following the program. Notice how the higher level statements of the program have been incorporated as comments.

```
100 REM ADMINISTER DRILL AND PRACTICE EXERCISE
110 REM VARIABLE TABLE
120 REM ANSWR$ : ANSWER TO ITEM
130 REM DATE$  : DATE
140 REM ITEM$  : ITEM
150 REM NITEM  : NUMBER OF ITEMS
160 REM NRESP  : NUMBER OF RESPONSES FOR ITEM
170 REM RESP$  : RESPONSE
180 REM SCORE  : STUDENT'S SCORE
190 REM SNAME$ : STUDENT'S NAME
200 REM
210 REM 1. INITIALIZE
220 REM    .1 GET FIRST ITEM
230                    READ ITEM$
240 REM    .2 SCORE = 0
250                    SCORE = 0
260 REM    .3 NITEM = 0
270                    NITEM = 0
280 REM    .4 OBTAIN STUDENT NAME AND DATE
290                    PRINT "INPUT YOUR NAME, PLEASE"
300                    INPUT SNAME$
310                    PRINT "INPUT TODAY'S DATE, PLEASE"
320                    INPUT DATE$
330 REM 2. ADMINISTER EXERCISE
340 REM    .1 WHILE ITEM <> "END"
350                    IF ITEM$ <> "END" THEN 370
360                    GOTO 670
```

```
370 REM      .1 GET ANSWER TO ITEM
380                    READ ANSWR
390 REM      .2 INITIALIZE AND PRESENT ITEM
400                    PRINT ITEM$
410                    NITEM = NITEM + 1
420 REM      .3 OBTAIN RESPONSE AND CHECK FOR RETRIES
430                    INPUT RESP$
440                    NRESP = 1
450 REM       .3 WHILE RESPONSE <> ANSWER TO ITEM
460 REM              AND NUMBER OF RESPONSES <3
470                    IF RESP$ <> ANSWR$ AND NRESP<3 THEN 490
480                    GOTO 530
490                    PRINT "TRY AGAIN"
500                    INPUT RESP$
510                    NRESP = NRESP + 1
520                    GOTO 450
530 REM       .4 CHECK FINAL RESPONSE AND DISPLAY FEEDBACK
540 REM         .1 IF RESPONSE = ANSWER TO ITEM
550                    IF RESP$ = ANSWR$ THEN 570
560                    GOTO 610
570 REM           .1 THEN
580                    PRINT "CORRECT"
590                    SCORE = SCORE + 4 - NRESP
600                    GOTO 640
610 REM           .2 ELSE
620                    PRINT "INCORRECT"
630                    PRINT "THE ANSWER IS " ; ANSWR$
640 REM       .5 GET NEXT ITEM
650                    READ ITEM$
660              GOTO 330
670 REM 3. OUTPUT RESULTS
680                    PRINT "YOUR SCORE IS ";SCORE;"/";NITEM*3
690                    LPRINT SNAME$,DATE$,SCORE
700              STOP
710     END
```

Notes:

a) LPRINT on line 690 outputs to the printer.

b) Microsoft BASIC allows identifiers up to 40 character in length.

c) The DATA statements are not included. They would follow the STOP at line 700

d) The interaction with the student and the output at the end is very minimal and in actual practice would need to be expanded.

Note the data dictionary at the beginning of the program. Documentation

of the variables, arrays and files used in a program is very important. Even when a programmer uses what seem to be very meaningful variable names when writing a program, the meaning of the names is not always apparent to everyone. In fact even the same programmer may have difficulty remembering the meaning of the names when returning to the program six months later. A useful convention is to group all variable names together, all array names together, and all file names together with each group arranged in alphabetical order. Each name should be followed by a brief explanation of its role in the program. This ordering facilitates quickly finding the meaning of any names used in the program.

Exercises

Translate the following into BASIC or whatever programming language you are using:

1)
```
IF A>B
    THEN T = A
      A = B
      B = T
```

2)
```
WHILE B>0
    A = A+1
    B = B-1
```

3) The guessing game of Chapter 5.

4) The sorting program of Chapter 6.

5) Modify the BASIC program of this section to include the modifications made in the exercises of Chapter 4.

9

Extensions to Structured Programming Statements

Although the three fundamental structured programming statements which we have been using are sufficient to program whatever a computer is capable of doing, there are certain situations in which these statements seem restrictive and labored. This section will consider two such situations which are frequently encountered and for which additional structured programming statements have been developed.

The first situation involves the use of nested selection statements. In some situations the nested levels appear to become unnecessarily difficult to follow. Consider the following example which assigns colors according to the value of a variable, N.

```
IF N=1
  THEN COLOR = "BLUE"
  ELSE IF N=2
    THEN COLOR = "RED"
    ELSE IF N=3
      THEN COLOR = "YELLOW"
      ELSE IF N=4
        THEN COLOR = "GREEN"
```

In this example the program seems to run off too far to the right and seems to require too many IF, THEN, ELSE words to simply check four values of a variable.

The CASE statement has been developed in order to test a sequence of conditions. It is written in the following format:

```
CASE OF
  (e1) s1
  (e2) s2
  (e3) s3
  ...
  (eN) sN
```

where e1 - eN are logical expressions, and s1 - sN are statements which may themselves be sequence, selection, repetition or case statements. The case statement is executed by testing the value of the first expression and if it is true, then the corresponding statement, s1, is executed. If e1 is not true, then e2 is tested, and if true, s2 is executed. This process continues until the first true expression is found in which case the corresponding statement is executed. After the corresponding statement has been executed, the execution of the case statement is complete.
If none of the expressions is true, then none of the statements are executed. Only one of the statements s1-sN will be executed for a given execution of the case statement. The nested selection example above can be written as the following case statement:

```
CASE OF
  (N=1) COLOR = "BLUE"
  (N=2) COLOR = "RED"
  (N=3) COLOR = "YELLOW"
  (N=4) COLOR = "GREEN"
```

The translation of this statement is the same as the translation of the nested selection, and the general pattern is:

```
        IF NOT(e1) THEN L1
        s1
        GOTO LN
L1      IF NOT(e2) THEN L2
        s2
        GOTO LN
L2      ...
        ...
        ...
LN
```

Following this pattern, the above example would translate into the following BASIC program:

```
100 IF N<>1 THEN 400
200 COLOR = "BLUE"
300 GOTO 1200
400 IF N<>2 THEN 700
500 COLOR = "RED"
600 GOTO 1200
700 IF N<>3 THEN 1000
800 COLOR = "YELLOW"
900 GOTO 1200
```

```
1000 IF N<>4 THEN 1200
1100 COLOR = "GREEN"
1200 ...
```

Pascal allows a version of a CASE statement which is somewhat more limited than that just described. In Pascal the selection within the CASE statement is according to the value of a variable or expression specified in the CASE header. Each alternative is preceded by a constant value. The value of the variable or expression is compared with the values preceding the alternatives. The first alternative thus matched is executed. If no match is found, then none of the alternatives is executed. The above example is written in Pascal as:

```
CASE N OF
    1: COLOR = 'BLUE'
    2: COLOR = 'RED'
    3: COLOR = 'YELLOW'
    4: COLOR = 'GREEN'
END;
```

In some cases we want a default or "otherwise" condition which is analogous to the ELSE part of the selection. In these situations we can write the last condition as "otherwise." Suppose for example that we want to assign "BLACK" to COLOR if N is other than 1 to 4. We can then write the CASE statement as

```
CASE OF
  (N=1) COLOR = "BLUE"
  (N=2) COLOR = "RED"
  (N=3) COLOR = "YELLOW"
  (N=4) COLOR = "GREEN"
  (OTHERWISE) COLOR = "BLACK"
```

We would then change lines 1000-1200 of the translation into BASIC to be

```
1000 IF N<>4 THEN 1150
1100 COLOR = "GREEN"
1125 GOTO 1200
1150 COLOR = "BLACK"
1200 ...
```

Pascal does not allow the OTHERWISE option.

As another example, consider writing the binary search program of Chapter 6 with a case statement as follows:

```
LO=1
HI=LIM
FOUND="FALSE"
WHILE HI>=LO AND FOUND<>"TRUE"
  MID=(HI+LO)/2
  CASE OF
    (KEY=A(MID)) FOUND="TRUE"
    (KEY<A(MID)) HI=MID-1
    (OTHERWISE) LO=MID+1
```

This last example illustrates a second sometimes cumbersome feature of the elementary structured programming constructs. In order to terminate the repetition when the value of KEY is found in array A, the logical variable, FOUND, is set and then tested as part of the termination condition. In a non-structured programming language, we would be likely to use a GOTO to exit from the body of the loop rather than set the value of FOUND to "TRUE." However, we have eliminated the GOTO because of the dangers described earlier. If we could limit the use of the GOTO to take us outside of the loop to the statement immediately following the repetition, then we would avoid the complexities and dangers of the general GOTO.

This leads to the second extension to the structured statements, the EXIT statement, which exits the enclosing repetition and continues execution with the statement immediately following that repetition. In the case of nested repetitions, the EXIT statement exits the innermost repetition in which the EXIT statement lies. Using the EXIT statement allows us to rewrite the above example as:

```
LO=1
HI=LIM
WHILE HI>=LO
  MID=(HI+LO)/2
  CASE OF
    (KEY=A(MID)) EXIT
    (KEY<A(MID)) HI=MID-1
    (OTHERWISE) LO=MID+1
```

In BASIC the translation of an EXIT is a GOTO to the line number of the statement immediately following the last statement of the repetition, which in the case of the WHILE is a GOTO back to the test of the repetition. Thus, the translation of this program fragment into BASIC is:

```
100 LO=1
200 HI=LIM
300 IF NOT(HI>=LO) THEN 1100
400 IF NOT(KEY=A(MID)) THEN 600
```

```
500 GOTO 1100
600 IF NOT(KEY<A(MID)) THEN 900
700 HI=MID-1
800 GOTO 1000
900 LO=MID+1
1000 GOTO 300
1100 _
```

Pascal does not allow the EXIT. It does, however, allow the GOTO.

It is important to note that neither the CASE nor EXIT statements compromise the principles of structured programming. The CASE statement can be reduced to a finite set of nested selection statements. The EXIT statement can be reduced to setting a variable which is tested in the termination condition of the repetition. Both of these statements preserve the "single entrance, single exit" characteristic of structured statements.

Exercises

1) Write a CASE statement which will assign a code from 1 to 7 to the variable NDAY corresponding to the day of the week in the variable DAY. For example if DAY="TUESDAY" then NDAY will be assigned 2.

2) The following segment of the drill-and-exercise program of Chapter 4 has a redundant test of RESPONSE=ANSWER.

```
REPEAT
  INCREMENT NUMBER ATTEMPTS
  OBTAIN RESPONSE
  IF RESPONSE=ANSWER
    THEN PRINT "CORRECT"
    ELSE PRINT "TRY AGAIN"
UNTIL RESPONSE=ANSWER OR NUMBER ATTEMPTS=3
```

That is, the test appears both within the body of the loop and within the termination condition.

Remove the test from the termination condition by using an EXIT within the body of the loop. Carry the modification into the translation of the program into BASIC in Chapter 8.

10

A Project : A Gradebook Program

One very common use of the computer in the classroom is to have the computer keep the teacher's gradebook. This section begins the development of a gradebook program as an example of a program which is larger than those which have been developed as examples earlier. It is also left for the reader to customize to his or her needs and tastes and to complete as a project in structured programming. A reasonable level of programming experience including the use of arrays and files is assumed.

The model for the gradebook is a manual gradebook which has a row for each student. Each row will contain the student name followed by a number of column entries for each assignment, quiz or exam score. Across the top of the gradebook the columns are labelled with column headings.

The operations which the program will have to provide will include creating or setting up the gradebook, editing the gradebook (which will include making new entries and altering entries already made), and displaying the gradebook. Since most computers will store the gradebook on disk or tape, two additional operations will be opening the gradebook and closing the gradebook. These operations will be selected by the use of commands. Once an operation has been selected, additional options can be specified by other commands specific to the particular operation.

We can now specify the first level of the program as:

```
ADMINISTER GRADEBOOK
  1. INITIALIZE
  2. REPEAT
    .1 OBTAIN COMMAND
    .2 CASE OF
      .1 (COMMAND = "CREATE") CREATE GRADEBOOK
      .2 (COMMAND = "OPEN") OPEN GRADEBOOK
      .3 (COMMAND = "CLOSE") CLOSE GRADEBOOK
      .4 (COMMAND = "EDIT") EDIT GRADEBOOK
      .5 (COMMAND = "DISPLAY") DISPLAY GRADEBOOK
    UNTIL COMMAND = "STOP"
  3. CHECK FOR OPEN GRADEBOOK
```

It should be obvious that the program will repeatedly accept commands until a stop command is entered. At the beginning we will want to perform whatever initialization is necessary, and at the end we will want to check to make certain that the gradebook has been properly saved. We can now proceed to the next level and consider each of the statements in turn.

The statement OBTAIN COMMAND can request and input commands. This might be implemented by use of a menu which displays the commands and allows the user to select the desired command or it might be implemented by simply asking the user to enter a command. More sophisticated programs might also incorporate a HELP command which would display instructions describing the use of the program. The refinement of this statement will be left to the reader.

To initially create the gradebook, it will be necessary to obtain the column headings and the student names. Also, depending on the particular disk or tape instructions for the computer you may be using, it may be useful to obtain a name for the gradebook. The statement at 2.2.1 can be refined into

2.2.1 CREATE GRADEBOOK
- .1 OBTAIN GRADEBOOK NAME
- .2 OBTAIN COLUMN HEADINGS
- .3 OBTAIN STUDENT NAMES
- .4 INITIALIZE GRADEBOOK

The statement 2.2.1.1 OBTAIN GRADEBOOK NAME can request the name from the user and then input the name. If the disk or tape system which is being used imposes certain restrictions on file names, the name input by the user should be checked to make certain it is a valid name. The refinement of this statement is left to the reader.

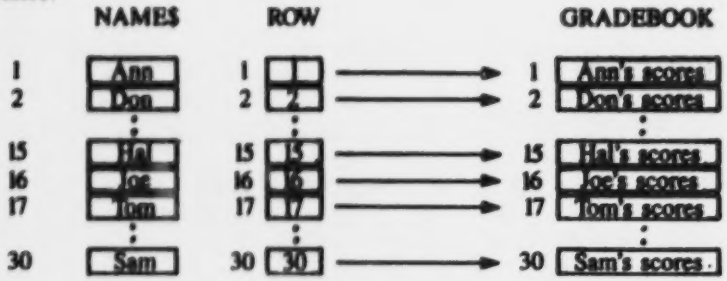
The statement 2.2.1.2 OBTAIN COLUMN HEADINGS will request the user to input the column headings. These can be stored in an array of strings. It will be necessary to either have the user specify the number of column headings or have the user indicate when the last heading has been entered. In the latter case the program will have to keep a count of the number. Again the refinement of this statement is straightforward and is left to the reader.

The statement 2.2.1.3 OBTAIN STUDENT NAMES can be refined in a manner very similar to that of the previous statement. The student names can be input into the elements of an array of strings. One significant difference is that the program may be expected to keep the names in alphabetical order. This is not difficult to do when the names are first entered. Rather than put the name directly into the array, the name can be input into a string variable. Then this name can be compared against the names already in the array, beginning with the first, until a name is found which comes after the given name. Then this latter name and all that follow can be moved down one place and the name just input inserted into the opening. Of course, the very first name input will be placed into the first element. This scheme works

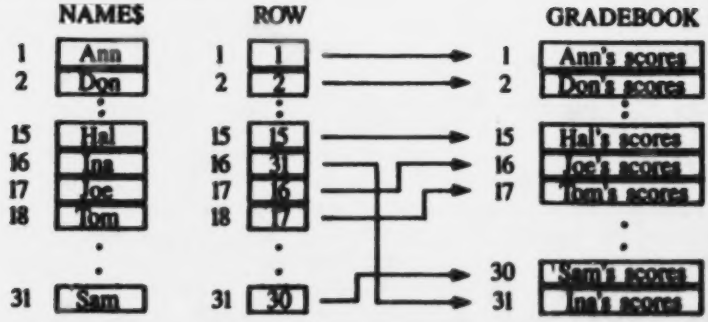
well when the names are first input. However, if the program allows names to be added to the gradebook at some later point in time, the ordering problem becomes more difficult. This is because a set of scores may have been entered for the students already in the gradebook. In this case to move the student names about would cause them to be placed out of order with respect to their scores. This means that we would either have to move all of a student's score when that student's name is moved, or we would have to keep some link tying a student's name to a particular row in the gradebook. The latter alternative is generally preferred. Following this scheme the scores in the gradebook would be kept in a two-dimensional array with one row for each student and one column for each heading. Then in addition to the string array containing student names we would have a second array which contains the number of the row for the scores of the particular student. Let's call the array of names, NAME$. Then the name of the tenth student is contained in NAME$(10). Let's call the array containing the numbers of the rows of the gradebook, ROW. Then the row in the gradebook for the tenth student will be contained in ROW(10).

When we first create the gradebook, there is no reason not to assign the first student to the first row, the second student to the second row, etc. This can be done by simply assigning the value 1 to ROW(1), the value 2 to ROW(2), etc. Now suppose that at some later point in time there are 30 names in the gradebook and that a name is added which fits between the 15th and the 16th names. This means that all of the names from the 16th to the 30th are moved down one place in the array NAME$. As we move these names down, we also move the 16th through the 30th elements of ROW down one place so that we can keep a link to the appropriate row in the gradebook. Then we insert the new name into NAME$(16) and assign ROW(16) the value 31 (since the 31st row is the next available row). Graphically this operation can be described as

Before:



After inserting the new name, Ina:



This scheme requires that a reference into GRADEBOOK be made with a subscript from ROW. For example, the index of Hal in the above example is 15. The row in GRADEBOOK is given by ROW(15). Then if we want to reference the 5th column of Hal's row we would write GRADEBOOK(ROW(15) , 5). The 7th column of Tom's row is written as GRADEBOOK(ROW(18) , 7) .

Using the scheme just discussed, we can refine 2.2.1.3 OBTAIN STUDENT NAMES as

```
2.2.1.3 OBTAIN STUDENT NAMES
        .1 OBTAIN NUMBER OF NAMES
        .2 NUMBER INPUT = 0
        .3 WHILE NUMBER INPUT < NUMBER OF NAMES
          .1 OBTAIN NEXT NAME
          .2 INDEX = 1
          .3 WHILE INDEX <= NUMBER INPUT
               AND NEXT NAME > NAME$(INDEX)
            .1 INDEX = INDEX + 1
          .4 IF INDEX <= NUMBER INPUT
            .1 TEMP INDX = NUMBER INPUT + 1
            .2 WHILE TEMP INDX >= INDEX
              .1 NAME$(TEMP INDEX) = NAME$(TEMP INDX+1)
              .2 TEMP INDX = TEMP INDX-1
          .5 NAME$(INDEX) = NEXT NAME
          .6 NUMBER INPUT = NUMBER INPUT + 1
        .4 INDEX = 1
        .5 WHILE INDEX <= NUMBER OF NAMES
          .1 ROW(INDEX) = INDEX
          .2 INDEX = INDEX + 1
```

The statement 2.2.1.4 INITIALIZE GRADEBOOK will set all of the

entries in the gradebook to either blank or a null value depending on whether gradebook information is kept as string data or as numeric data. The advantage of keeping string data is that letter grades can be entered and that blanks can be used for spaces in which entries have not yet been made. The advantage of keeping numeric data is that weighted sums and other computations can be made. If numeric data is kept, then some value must be chosen for a space in which an entry has not yet been made. If zero scores can be entered as valid scores, zero may not work well for this purpose. Some particular negative value might be used. This statement will also set the gradebook to "open" status. The refinement of this statement is left to the reader.

The statement 2.2.2 OPEN GRADEBOOK should first test whether or not a gradebook is already open and if so, indicate an error. If no gradebook is open, then this statement should obtain a gradebook name and then input from disk or tape all of the data pertaining to that gradebook. This refinement could be written as

```
2.2.2 OPEN GRADEBOOK
      .1 IF GRADEBOOK OPEN
        THEN
          .1 DISPLAY OPEN MESSAGE
        ELSE
          .1 OBTAIN NAME
          .2 OPEN FILE
          .3 INPUT COLUMN HEADINGS
          .4 INPUT STUDENT NAMES AND ROW INDICES
          .5 INPUT GRADES
          .6 SET GRADEBOOK OPEN
```

As this refinement indicates all of the information will have been written in the file on disk or tape. This would have been done by a close gradebook operation. In addition to the information contained in arrays, the number of elements in each array should be written just before the array itself in order to allow reading in the appropriate number of elements. Further refinement of this statement is left to the reader.

The refinement of 2.2.3 CLOSE GRADEBOOK is just the reverse of the last statement. It will first test that a gradebook is open. If one is open, the statement will then write all of the information to a file and then set the gradebook closed. This refinement is left to the reader.

The edit operation will incorporate several options. These will include as a minimal set adding a column, adding a name, changing a column heading, changing a name, deleting a column, deleting a name, entering a grade, changing a grade, and exiting or returning to the outer repetition of statement 2. This operation should also test to make certain that a

gradebook is open before requesting options. This refinement can be written as

```
2.2.4 EDIT GRADEBOOK
      .1 IF GRADEBOOK OPEN
        THEN REPEAT
          .1 OBTAIN OPTION
          .2 CASE OF
            .1 (OPTION = "ADD COL") ADD COLUMN
            .2 (OPTION = "ADD NAME") ADD NAME
            .3 (OPTION = "CHANGE COL") CHANGE COLUMN
            .4 (OPTION = "CHANGE NAME") CHANGE NAME
            .5 (OPTION = "DEL COL") DELETE COLUMN
            .6 (OPTION = "DEL NAME") DELETE NAME
            .7 (OPTION = "ENTER GRADE") ENTER GRADE
            .8 (OPTION = "CHANGE GRADE") CHANGE GRADE
          UNTIL OPTION = "EXIT"
        ELSE DISPLAY "GRADEBOOK NOT OPEN"
```

In light of the previous discussion of the data structure of the gradebook, these options are straightforward. The statement ENTER GRADE may have further options such as entering a single grade for a single student, entering a single grade for all students, entering a series of grades for a single student, and entering a series of grades for all students. To give some illustration of how entries are located in the gradebook, a simple refinement of ENTER GRADE is given as:

```
2.2.4.1.2.7 ENTER GRADE
            .1 OBTAIN OPTION
            .2 CASE OF
              .1(OPTION = "CLASS") ENTER CLASS GRADE
              .2(OPTION = "STUDENT") ENTER STUDENT GRADE
```

Then a simple refinement of ENTER STUDENT GRADE is:

```
2.2.4.1.2.7.2.2 ENTER STUDENT GRADE
                .1 OBTAIN STUDENT NAME
                .2 SEARCH FOR STUDENT INDEX
                  .1 STDNT IX = 1
                  .2 WHILE STDNT IX <= NUM STDNT
                    .1 IF STUDENT NAME = NAME$(STDNT IX)
                      .1 THEN EXIT
                    .2 STDNT IX = STDNT IX + 1
                  .3 IF STDNT IX <= NUM STDNT
                    .1 THEN
```

```
.1 OBTAIN COL HEAD
.2 COL IX = 1
.3 WHILE COL IX <= NUM COL
  .1 IF COL HEAD = COL$(COL IX)
    .1 THEN EXIT
  .2 COL IX = COL IX + 1
.4 IF COL IX <= NUM COL
  .1 THEN
    .1 OBTAIN GRADE
    .2 GRADEBOOK(ROW(STDNT IX),
          COL IX) = GRADE
  .2 ELSE DISPLAY "BAD COL"
.2 ELSE DISPLAY "CAN'T FIND STUDENT"
```

The statement 2.2.5 DISPLAY GRADEBOOK will also have options. These should include displaying grades for a single student, displaying a column for all students, or displaying the entire gradebook. Options within these could be to display to the screen or the printer. The refinement of this statement can be written as

```
2.2.5 DISPLAY GRADEBOOK
    .1 IF GRADEBOOK OPEN
      .1 THEN REPEAT
        .1 OBTAIN OPTION
        .2 CASE OF
          .1 (OPTION = "STUDENT") DISPLAY SINGLE STUDENT
          .2 (OPTION = "COL") DISPLAY SINGLE COLUMN
          .3 (OPTION = "ALL") DISPLAY ALL
        UNTIL OPTION = "EXIT"
      .2 ELSE DISPLAY "GRADEBOOK NOT OPEN"
```

In light of the previous discussion, further refinement of these statements should be straightforward.

Many other features could be incorporated into the gradebook program. What has been developed has been designed to provide a structure within which other features can be designed. The reader is encouraged to customize the gradebook to fit his or her needs, tastes and computer system.

Appendix: Proving That Statements Are Correct

The reason for restricting the programming statements is not to make programming more difficult or strenuous, but, as was stated in the introduction, to limit ourselves to programming statements which can be thoroughly understood. In Dijkstra's words these are "intellectually manageable." To thoroughly understand a program, we must be able to convince ourselves, using whatever degree of rigor is required, that the programs written by us will in fact perform the intended calculations. This might mean in the extreme case that we would want to present a formal proof of the program. It is quite possible in most programming languages, such as BASIC, to write programs which have such a complex or contorted control structure that we have no systematic strategy for proving assertions about the programs. The kinds of assertions we want to prove have to do with the computation that the program invokes in a computer. We generally want to convince ourselves that the computations in fact do agree with computations set forth in the solution designed for the problem at hand or with the computations described in the program specifications.

This appendix will discuss strategies for constructing proofs of assertions regarding the behavior of programs. Although most of us will not construct formal proofs of our programs, having some insight into the strategies for doing so will help us reason informally to convince ourselves that our programs will work. Also, having some insight into these strategies will help us write our programs in such a way that reasoning about them will be made easier. This will lead to programs that are easier to understand.

In order to talk about the computation invoked by a program, we can refer to the transformation of the input data produced by the program. This can be concisely described as a function of the input data. In other words, we can consider any program as starting with certain values for its variables, input files and DATA statements and then transforming these values to the final variable values and the output values of when the program has completed execution.

For those readers who consider themselves non-mathematically inclined, the use of functional concepts may seem to be an unnecessary excursion into obscurity. There is, however, real value in making use of mathematical

clarity if we are going to talk unambiguously about the computation invoked by a program. In general a function can be thought of as a way of relating two values. One of the values results from the application of the function to the other value. Thus functions can be defined by pairs of values. We also say that the function transforms one value into another value. In everyday language we might say that someone's sense of security is related to the size of his bank account. If we could quantify both sense of security and the size of a bank account, then we might describe a function such that for any given value of the bank account we would know the value of the sense of security. We would then be able to say that sense of security was a function of the size of the bank account. Of course, sense of security is normally related to many other factors in a person's life, and we would have to talk about a person's state of being. In order to describe a person's state of being, we might consider his sense of security, happiness, confidence, bank account, primary relationship, and so on. Then a change in the bank account would result in a change in the whole state of being.

Similarly, we will talk about the state of the computation as the set of values of all of the data entities which have been defined for a program. The computation performed by any statement in the program can then be described as a function which, when applied to any given state of the computation, will give us a new or transformed state of the computation. If our program has defined the variables A,B,C,XX,SUM,TIME and IX, then the state of the computation would be the set of values for these variables. Since in general when defining a function on these variables we do not know their values, we use the variable names to represent the values, and we write the state of the computation as the list of variables enclosed in parentheses, as:

(A,B,C,XX,SUM,TIME,IX) .

Since the values of the variables may change as a result of the function invoked by a statement, it is often useful to have some notation which will keep track of the values of the variables before and after the execution of the statement. We will write the variables with a subscript. For example *A0* will indicate the value of variable A at "time 0" or before the execution of a statement, and A1 will be the value at the next "time period" or after execution of the statement. The assignment statement

A = B + C

represents a function transforming (A0,B0,C0) into (A1,B1,C1). We can write this as

F(A0,B0,C0) = ((A1,B1,C1) <--(B0+C0,B0,C0))

This can be read as "the function, F, of the state (A0,B0,C0) is defined

as the transformation which gives A1 the value of B0+C0, B1 the value of B0, and C1 the value of C0. This indicates that the function transforms the value of A to the sum of B and C and leaves the values of B and C unchanged. When the values of the variables before and after execution will not be confused, the subscripts may be omitted and the function will appear as:

F(A,B,C) = (A,B,C) <-(B+C,B,C)

By using this sort of functional notation we can keep track of variables which do not change value as well as those which are changed by the statement.

While an assertion that the value of A is transformed to the sum of B and C is clearly obvious from examining the statement, other kinds of assertions can also be made. For example, if both B and C have positive values before the execution of the statement, then following the execution of the statement A will be larger than either of B or C. More formally:

(B0>0 and C0>0) => (A1>B1 and A1>C1)

where => is read as "implies."

Although the function of any statement is a function of the state of the program, since a given statement will reference only some of the variables, its function will normally be written only as a function of the variables referenced and the other values in the state will be understood to remain unchanged. Of course this may occasionally lull us into overlooking some changes in values. For complete certainty we would have to explicitly write all of the assumed results of the statement.

In order to convince ourselves that a given statement performs the computation which we want it to compute, we will work with three sets of information: 1) a set of assertions, called *pre-assertions*, which describe the state of the computation prior to execution of the given statement, 2) a definition of the function of the statement, 3) a set of assertions, called *post-assertions*, describing the state of the computation following the execution of the statement. If we can prove to ourselves that the application of the function to the state of the computation prior to the execution will produce the assumed state following the execution, then we will have convinced ourselves that the statement indeed does what we want it to. This may seem convoluted, tautological, and little more than confusing, but the process is meaningful. It is important to realize that the function is derived from the statement written in the programming language, while the assertions concerning the state of the computation are derived from our understanding of the problem, and from the specifications for the program. From our specifications of the problem we know that we want the statement to change the values of one or more of the variables in the state of the computation.

From the specifications we can derive appropriate pre-assertions regarding the state of the computation prior to execution of the statement and appropriate post-assertions regarding the desired state after execution of the statement. Given the statement in the programming language, we can derive its function. Then, if we can show that the post-assertions result from applying the function to the state as described in the pre-assertions, we know that the statements evoke the desired computation. We can show this as follows:

[assertions regarding prior state]

STATEMENT [function derived from statement]

(proof of statement)

[assertions regarding post state]

The actual inclusion of the pre- and post-assertions as comments in a program can, in many cases, provide important aids for understanding what a program does. It is important to emphasize that in some cases our focus upon certain variables may cause us to overlook the effect of the computation on other variables, and this "secondary" effect may lead to errors. For example, in focusing on the exchange of values of A and B in the following, we may leave T out of the description of the function and write the function as (A,B) <-- (B,A).

```
100 T = A
200 A = B
300 B = T
```

If T were being used in some larger sequence or loop containing this sequence, the alteration of T could introduce an error. It is important to write the function as

(A1,B1,T1) <-- (B0,A0,A0) .

Then if our post-assertion includes an assertion that T is unchanged, the function would not satisfy our needs.

Proofs for Sequences

When we attempt to convince ourselves that a sequence of statements performs a desired computation, we begin with the pre- and post-assertions of the sequence taken as a single statement and the pre- and post-assertions of each statement within the sequence. The pre-assertions for the sequence should be sufficient to derive the pre-assertions of the first statement, the post-assertions of the first statement together with the pre-assertions of the sequence should be sufficient to derive the pre-assertions of the second statement, and so on until the post-assertions of the last statement together with the pre-assertions of the sequence and the preceeding post-assertions should be sufficient to derive the post-assertions of the sequence. We assume that the transformation of the state for each given statement within the sequence is such that we can prove that the post-assertions for that statement can be derived given the pre-assertions for that statement. For a three-statement sequence we can illustrate this as:

A. [sequence pre-assertions]
  B. [statement 1 pre-assertions]
    statement 1
  C. [statement 1 post-assertions]
  D. [statement 2 pre-assertions]
    statement 2
  E. [statement 2 post-assertions]
  F. [statement 3 pre-assertions]
    statement 3
  G. [statement 3 post-assertions]
H. [sequence post-assertions]

We must be able to derive the pre-assertions for a given statement within the sequence from the sequence pre-assertions and the post-assertions for the prior statements in the sequence. Finally, we must be able to prove that the sequence post-assertions follow from the sequence pre-assertions and all of the statement post-assertions. Then reading [A,B] as "assertions A together with assertions B" and reading [A] –> [B] as "assertions B can be derived from assertions A" and reading [A] -F-> [B] as "assertions B can be derived from assertions A and the computation of F", we can show the proof sequence of the above example as:

1. [A] –> [B]
2. [B] -F1-> [C]
3. [A,C] –> [D]
4. [D] -F2-> [E]
5. [A,C,E] –> [F]
6. [A,F] -F3-> [G]
7. [A,C,E,G] –> [H]

When we convince ourselves that these steps can be proved, we then know that the post-sequence assertions, H, can be derived from the pre-sequence assertions, A, and the function for the sequence of statements. If the functions of the statements are Fl, F2, F3, then the function of the sequence is the composite function F(X) = F3(F2(Fl(X))) where X is the state of the computation prior to execution of the sequence. This composition can be understood by observing that F3 has as its arguments the state as transformed by F2 which has as its arguments the state as transformed by Fl.

As an example, consider a simulation of a spaceship traveling at a constant velocity on a numbered grid. Knowing that the spaceship has taken Tl units of time to travel from point (X1,Y1) to point (X2,Y2), we want to compute how long it will take to travel from (X2,Y2) to (X3,Y3). The following program segment provides a straightforward computation of the time.

```
D1 = SQRT( (X1-X2)**2+(Y1-Y2)**2 )
V = D1/T1
D2 = SQRT( (X2-X3)**2+(Y2-Y3)**2 )
T2 = D2/V
```

We can argue that T2 is the time required to travel from (X2,Y2) to (X3,Y3) with the following assertions and functions.

A1.[The simulated spaceship requires T1 units of time to travel from (X1,Y1) to (X2,Y2). (X3,Y3) is another point in space.]

F1. (D1,X1,Y1,X2,Y2) <- (SQRT((X1-X2)²+(Y1-Y2)²2,X1,Y1,X2,Y2)

R1.[D1 is the distance between (X1,Y1),(X2,Y2)]

F2. (V,D1,T1) <- (D1/T1,D1,T1)

R2.[V is the velocity of the spaceship.]

F3. (D2,X2,Y2,X3,Y3) <- (SQRT((X2-X3)²+(Y2-Y3)²,X2,Y2,X3,Y3)

R3.[D2 is the distance between (X2,Y2),(X3,Y3).]

F4. (T2,D2,V) <- (D2/V,D2,V)

R4.[T2 is the time required to travel from (X2,Y2),(X3,Y3).]

A1. is the sequence pre-assertion. R1,R2,R3,R4 are the post-assertions for the corresponding statements. R1 makes use of the sequence pre-assertion and the function. R2 makes use of the sequence pre-assertion regarding the

value of T1 and of the information from R1. R3 makes use of the sequence pre-assertion. R4 makes use of the information from R2 and from R3. The final post-assertion, R4, is also the sequence post-assertion which was to be proved.

In the example just given, the function of each statement was independent of the functions of the other statements in the sense that the variables referenced and assigned values by a given statement were neither referenced nor assigned values by any of the other statements.

In some instances a sequence of statements s1,s2,s3 will be interdependent. In such instances we may want to derive the function computed by the sequence as a whole. For example consider the following sequence of statements:

```
100 X = Y + Z
200 Y = X + Z
300 Z = Y + X
```

Although we can easily write the functions of the individual statements as

```
(X1,Y1,Z1) <- (Y0+Z0,Y0,Z0)
(X1,Y1,Z1) <- (X0,X0+Z0,Z0)
(X1,Y1,Z1) <- (X0,Y0,Y0+X0)
```

respectively, the composite function for the sequence may not be immediately obvious, and we want a systematic way of deriving this function. A systematic approach is to trace the effect of each statement on the state of the program which in this case consists of the three variables X,Y,Z. In order to trace the action we can denote the original values of X,Y,Z as X0,Y0,Z0, the values after the execution of the first statement as X1,Y1,Z1 the values after execution of the second statement as X2,Y2,Z2 and so on. Then we can list the states of the computation in a table as follows:

statement	X	Y	Z
100	Y0+Z0	Y0	Z0
200	X1	X1+Z1	Z1
300	X2	Y2	Y2+X2

Then by working backwards up each column of the table, we can derive the effect of the sequence on each variable. Thus,

```
X3 = X2          Y3 = Y2          Z3 = Y2+X2
   = X1             = X1+Z1          = X1+Z1+X1
   = Y0+Z0          = Y0+Z0+Z0       = Y0+Z0+Z0+Y0+Z0
                    = Y0+2*Z0        = 2*Y0+3*Z0
```

Then the composite function can be written as:

(X1,Y1,Z1) <- (Y0+Z0,Y0+2*Z0,2*Y0+3*Z0)

This tells us that X1 will be replaced by the value of Y0+Z0 which is obtained from the first column of equations, that Y1 will be replaced by Y0+2*Z0 which is obtained from the middle column of equations and that Z1 will be replaced by 2*Y0+3*Z0 which is obtained from the rightmost column of equations.

Proofs for Selection Statements

Given the selection statement

IF e THEN s1 ELSE s2

we can write the function for s1 as F1 and the function for s2 as F2. Of course if these were sequence statements we might have derived the functions by a trace of the state as shown above. In any case we will write the function, F, of the selection statement as:

F(X) = (X) <- (e -> F1(X), NOT(e) -> F2(X))

where X stands for the state of the computation.This notation makes use of the conditional symbol ->, and we can read the selection function as "if e then F1(X) and if not e then F2(X)."

Another notation for a conditional function, F(X), is

```
F(X) = if e then F1(X)
          if NOT(e) then F2(X)
```

Although the first notation is more concise, in some cases it may be clearer to list the alternatives on separate lines.
Given the selection statement

```
IF A<B THEN A = B+C
       ELSE A = D+E
```

we can write the function as:

```
F(X) = if A<B then (A,B,C) <- (B+C,B,C)
          if A>=B then (A,D,E) <- (D+E,D,E)
```

In order to determine whether or not the function of a selection statement does what we want it to, we must be able to prove that the post-assertions follow from the pre-assertions together with the conditions for the first alternative and its function, and we must be able to prove that the post-assertions follow from the pre-assertions together with the conditions for the second alternative and its function. Consider the following example:

```
pre-assertion:
  [ A,B have numeric values ]

program:
  IF A <= B
    THEN LO=A
      HI=B
    ELSE LO=B
      HI=A

post-assertion:
  [ LO<=HI ]
```

program function:

F(X) = if A<=B then (A,B,LO,HI) ← (A,B,A,B)
if A>B then (A,B,LO,HI) ← (A,B,B,A)

In order to show that the post-assertion follows, we must consider the case where A< B and the case where A>B.

If A<=B then LO=A and HI=B which implies LO<=HI.
If A>B then LO=B and HI=A which implies LO<=HI.

Thus to prove that a selection statement is correct, we prove that the function performed under each of the possible alternative conditions does in fact perform the desired computation.

The proof of a more complex nested selection statement requires proving that all of the possible combinations of conditions lead to the correct computation. This requires deriving the functions for each of the combinations of conditions. A more complex nested selection statement is given below. Indentation and labeling have been used to offset the conditions and alternatives at each level. The THEN and ELSE words are indented from the corresponding IF, and sequences of alternatives are indented from the THEN or ELSE words. The function of this nested statement is to assign to LO, MID and HI the lowest, middle and highest values respectively of A,B and C.

```
IF A<=B THEN
    [S1] IF C<=A THEN
        [S1.1] LO=C
               MID=A
               HI=B
      ELSE
        [S1.2] IF C<=B THEN
               [S1.2.1] LO=A
                        MID=C
                        HI=B
             ELSE
               [S1.2.2] LO=A
                        MID=B
                        HI=C
  ELSE
    [S2] IF C<=B THEN
        [S2.1] LO=C
               MID=B
               HI=A
      ELSE
        [S2.2] IF C<=A THEN
               [S2.2.1] LO=B
                        MID=C
                        HI=A
             ELSE
               [S2.2.2] LO=B
                        MID=A
                        HI=C
```

Then systematically following through the conditions, we arrive at the following six possible combinations together with the corresponding functions: (Subscripts have been dropped from the variable names for brevity.)

If A<=B and C<=A then (A,B,C,LO,MID,HI) <- (A,B,C,C,A,B)
If A<=B and C>A and C<=B then (A,B,C,LO,MID,HI) <- (A,B,C,A,C,B)
If A<=B and C>A and C>B then (A,B,C,LO,MID,HI) <- (A,B,C,A,B,C)
If A>B and C<=B then (A,B,C,LO,MID,HI) <- (A,B,C,C,B,A)
If A>B and C>B and C<=A then (A,B,C,LO,MID,HI) <- (A,B,C,B,C,A)
If A>B and C>B and C>A then (A,B,C,LO,MID,HI) <- (A,B,C,B,A,C)

From this function description we could prove the post-assertion that LO<=MID<=HI. This is accomplished by proving that

LO<=MID<=HI for each of the six possible cases above. For example consider the third case:

If A<=B and C>A and C>B then
    LO has value A
    MID has value B
    HI has value C
Then A<=B implies LO<=MID
        C>A implies HI>LO
        C>B implies HI>MID and MID<HI

Thus LO<=MID<HI
and LO<=MID<=HI which is what we want to prove. The reader is invited to prove the assertions for the remaining five cases.

Proving Repetition Statements

Understanding the computation invoked by a sequence statement or a selection statement follows directly from looking at the program statements. In the case of a sequence statement, we know that the computation consists of the computation invoked by the first statement followed by that invoked by the second statement followed by that of the third and so on. In the case of a selection statement, we know that given a certain state of the computation that a certain alternative will be selected and that the computation consists of the computation invoked by that alternative. However, in the case of a repetition statement, the actual computation will consist of some number of iterations of the repeated statement, and exactly how many iterations is not always easily determined by examining the state of the computation immediately prior to the execution of the repetition statement. In order to prove some assertion regarding the repetition statement we must be able to present a general argument which will hold for any number of repetitions. We have two approaches for making such arguments — one based upon the notion of mathematical induction and the other based upon the notion of a *loop invariant*. Mathematical induction is a process we can use to argue about properties of natural numbers (e.g. 1,2,3,...). This process is based upon the fact that if a property can be shown to be true for 1, and if by assuming that the property holds for any number N we can then show that the property is true for N+1, then the property holds true for all of the natural numbers.

Each of these approaches will be described for the following very simple program which computes the sum of two numbers.

```
WHILE B>0
  A = A+1
  B = B-1
```

The pre-assertions for this program are: A and B are defined numbers, B is an integer, and B>0 . The post-assertion for the program is [A = A0+B0] (That is the final value of A is equal to the sum of the initial values of A and B).

Using an induction argument, we argue that if the program computes A+B correctly for B=1 and if by assuming that the program works correctly for B=N, we can then show that it works correctly for B=N+1, then we know that it will work for all values of B taken from the natural numbers. This is a two-step process consisting of

1) showing the program works for B=1
2) Given that the program works for B=N, then showing that the program works for B=N+1

1) For B=1, on entry B>0 and the body of the loop is executed. A is then assigned A0+1 and B is assigned B0-1 which is equal to 0. The termination condition is then met and A = A0+1 which is equal to A0+B0. Therefore, the post-assertiom holds.

2) Assume that the post-assertion holds for B=N. Then consider B = (N+1). On entry B>0 and A is assigned A+1 and B is assigned B-1 which is equal to (N+1)-1 or N. Then A has value A0+1 and B has value N. Then we assumed that when B=N the loop computes A+B which is (A0+1)+N which equals A0+(N+1) which is A0+B0. We then know that the post-assertion holds for all positive values of B.

Induction arguments are sometimes useful when proving that a program works for an array of any length or for a string of any length. However, a more elegant argument can often be made using loop invariants. A *loop invariant* is an assertion which can be shown to be true at the beginning of the execution of the body of a repetition statement and to be true after the execution of the body of the loop. Even though execution of the body will change the state of the computation, the loop invariant is an assertion devised in such a way that it remains true. Although there are some formal methods of deriving loop invariants, when we are the authors of a program we know what we expect a given repetition to compute, and we will then generate candidate assertions in such a way as to reflect the expected computation. Given a candidate assertion, we then have to show that the assertion is true upon entry to the repetition statement and that it is true after the execution of the body of the repetition. In the above example we can observe that the assertion

A+B = A0+B0

is a loop invariant. On entry to the loop, A = A0 and B = B0. Therefore, A+B = A0+B0. After execution of the body of the loop, A = A0+1 and B = B0-1 . Therefore, A+B = A0+1+B0-1 = A0+B0. By this last

argument, we see that the assertion will hold before and after every execution of the body of the loop, and thus the assertion is a loop invariant.

In general loop invariants are assertions regarding constant or invariant relationships among variables which themselves change during the execution of the loop. Thus in the above example A gets larger while B gets smaller, and their sum remains constant. However, this is a feature of the relationship between A and B and does not tell us anything about the actual value of A or the actual value of B. In order to assert something about the value of A or B we must take into account the termination condition of the repetition. It is at this point that the power of the loop invariant arguments become apparent. In this example the loop will terminate when B=0. After termination of the loop, we know that A+B = A0+B0 and that B=0. Therefore, A+B=A+0=A=A0+B0 which is what we want to prove about the program.

In order to make use of the termination condition we must of course be able to prove that the termination condition will in fact be reached. In many cases such as the above example, the termination condition involves some value which decreases or increases regularly with each execution of the loop until some limit is reached. In the above example we know that B is initially equal to a positive integer and that after each execution of the loop that B1 =(B0-1) so that B1<B0 and therefore B will eventually meet the termination condition.

The strategy for proving assertions regarding repetition statements using loop invariants will then consist of three steps:

1) show that the termination condition will be met
2) find an appropriate loop invariant which must be shown to hold on first entry to the loop and to hold after each execution of the loop
3) show that the assertion follows from the termination condition together with the loop invariant.

As another example consider the following program which computes the factorial of a number.
(The factorial of N, written N!, is defined for integers >=0 as:
if N=0 then N!=1
N>0 then N!=1*2*3*...*N)

```
FACT = 1
WHILE NUM > 1
  FACT = FACT * NUM
NUM = NUM-1
```

Again, we will first use an induction argument and then a loop invariant argument. The induction argument consists of arguing that if the program computes NUM! correctly for NUM=1 and if by assuming that the program

works correctly for NUM=N we can then show that it works correctly for NUM=N+1, then we know that it will work for all values of NUM taken from the natural numbers. In order to actually carry out the proof, it will be easier to first look at the function of the while statement which computes FACT*(NUM!) and then observe that FACT=1 prior to the execution of the while.

1) The program computes FACT*(NUM!) for NUM=1 since the body of the while will never be executed and FACT will be unchanged and hence equal 1.

2) Assume that the while computes FACT*(NUM!) for NUM=N. Then if NUM=N+1, after the first execution of the body of the while, FACT equals FACT*(N+1) and NUM=N. The remaining repetitions will then compute FACT*(NUM!) by assumption and this is FACT*N!*(N+1) which is FACT*((N+1)!).

Then observing that FACT=1 prior to execution of the repetition, we have FACT = 1 * (N+1)! which is what we set out to prove.
In order to use a loop invariant argument, we can observe that the assertion

FACT*(NUM!) = N!

where N is the value of NUM, is a loop invariant. We can see this by observing that on first entry FACT has value 1 and NUM has value N. Then FACT*(NUM!) is the same as 1*(N!) which is N!. After execution of the body of the repetition, FACT is equal to FACT*NUM and NUM now has the value NUM-1. Then
(FACT*NUM)*((NUM-1)!) is the same as FACT*(NUM!) which before execution of the body was equal to N!.The loop will terminate when NUM=1. After termination of the while statement we know that FACT*(NUM!)=N! and that NUM=1. Therefore FACT*(NUM!)=FACT*1=N!, and we then know that FACT=N!, which is what we want to prove about the program.

In order to prove that the termination condition will in fact be reached, we know that NUM is initially equal to a positive integer, N, and that after each execution of the loop that NUM1 = (NUM0-1) so that NUM1 < NUM0 and therefore NUM will eventually meet the termination condition.

Working with loop invariants may seem awkward or mystifying at first and may require a slightly different style of thinking about loops. One way of thinking about loops which may help is to consider them as iterative processes analogous to numerical methods with use iterative computations to converge upon a solution. Such a process iteratively narrows the ranges of possibilities in which a solution may be found. In an analogous manner a loop searching a list of values iteratively reduces the length of the list, or a string processing loop may iteratively reduce the length of a substring

yet to be processed. In the above example, the loop reduced NUM which represented the number of values yet to be multiplied in computing the factorial. From this viewpoint the loop invariant will describe a relationship between the solution (or value to be computed, or whatever is being asserted in the assertion statement) and the "space" yet to be processed.

As an example of this approach consider the following binary search program which searches an array, A, of length LIM of values arranged in ascending order for a match with the value in KEY. (This is of course a variant of the guessing game example presented earlier.)

```
LO = 1
HI = LIM
FOUND = "FALSE"
WHILE HI>=LO AND FOUND<>"TRUE"
  MID = INT((HI+LO)/2)
  IF KEY = A(MID)
    THEN FOUND = "TRUE"
    ELSE
      IF KEY < A(MID)
        THEN HI=MID-1
        ELSE LO=MID+1
```

Each successive execution of the loop halves the length of the list which can contain the value of KEY until either the value is found or until no more of the list remains to be searched. While we cannot tell what the final values of HI and LO will be, we do know that the difference between them is approximately halved with each execution of the loop. We can show that after each execution of the loop that

(HI1-LO1) < (HI0-LO0)

and that eventually HI-LO will be less than 0 which implies that HI<LO which is sufficient to terminate the loop. The other possible termination condition is that the value of KEY is found in the list and that FOUND will be set to "TRUE."

In order to find the appropriate loop invariant, we can begin with what we want the program to do — search the list. Two possibilities can occur: first that the value of KEY will be found in which case A(MID) will contain that value and FOUND will be equal to "TRUE", and second that the value of KEY is not in the list which we know by having considered all elements of the list. The first possibility can be stated as:

IF FOUND = "TRUE" THEN KEY = A(MID)

Since the program does not actually examine all of the list elements but eliminates halves of the remainder at each step, we cannot assert that all

of the elements have been tested if the loop terminates without KEY having been matched. Instead we must observe that at each step if KEY is to be found in the portion of the list yet to be tested that it will be found in the portion of the array between and including A(LO) and A(HI). This possibility can be expressed as:

IF FOUND = "FALSE" THEN (IF KEY=A(J) THEN LO <= J <= HI)

Now we can observe that if HI<LO, which is one of the termination conditions, then there is no J such that LO <= J <= HI and therefore KEY is not equal to any element of the list. These two assertions can be combined into the loop invariant. On first entry to the loop, the value of FOUND is "FALSE" and if LIM > 0 then LO <= HI. Therefore, if KEY is to be found in the list, it will occur between and including A(LO) and A(HI) which on first entry is the entire list. After execution of the body of the loop, either FOUND has value "TRUE" and KEY is equal to A(MID) or KEY has not been found and the list has been reduced to the half which may still contain the value of KEY. We were able to eliminate half of the list since we know the list is sorted into ascending order, and we know that either KEY < A(MID) or KEY > A(MID). Thus we have the loop invariant.

(IF FOUND="TRUE" THEN KEY = A(MID)) AND
(IF FOUND="FALSE" THEN (IF KEY = A(J) THEN LO <= J <= HI))

together with the termination condition

HI<LO OR FOUND "TRUE"

From this we know that either FOUND will be equal to "TRUE" and KEY will equal A(MID) or that KEY does not occur in the list. This is what we want the program to do.

Another, slightly different, way of thinking about loops which may help a programmer derive appropriate loop invariants is to view the loop as "consuming" one space and constructing another "space." In the factorial example above, we can view the loop as consuming the value of NUM. The value of NUM becomes smaller with each iteration of the loop. At the same time the value of FACT is being increased. We can then look for a constant relationship between these mutually changing values and discover that FACT*(NUM!)=N!, which is the loop invariant.

As another example of proving repetition statements using loop invariants, consider the bubble sort program presented in Chapter 6. as:

```
TOP = LIM
WHILE TOP > 1
  I=1
  WHILE I<TOP
    IF A(I) > A(I+1)
      THEN T=A(I)
        A(I)=A(I+1)
        A(I+1)=T
    I=I+1
  TOP=TOP-1
```

This program "bubbles" the largest element in the remaining lower part of the array to the top of this remaining part. By "lower part" of the array we mean those elements whose subscripts are lesser in value. By "top" we mean the element with the largest subscript. The first time through the loop the largest value is moved to the top. The next time through the next largest value is moved to the next to the top location and so on. At any point in the computation the values above A(TOP) are in order and all of the values below A(TOP) are less than or equal the values above A(TOP). By elements above and below is meant elements with subscripts greater than and less than respectively. In the inner loop, I is the index of the largest value so far as the index increases. If the element, A(I), has a value greater than the element immediately above it, A(I+1), then the larger value is bubbled up by exchanging it with the value in A(I+1). In this fashion we observe that all values of elements below A(I) are less than or equal to the value of A(I). These observations lead to the loop invariants inserted in the program. The obvious termination conditions have also been inserted.

```
TOP=LIM
WHILE TOP>1
  {EVERYTHING ABOVE A(TOP) IS IN ORDER AND EVERYTHING
  BELOW AND INCLUDING A(TOP) IS LESS THAN OR EQUAL TO
  EVERYTHING ABOVE A(TOP)}
  I=1
  WHILE I<TOP
    {EVERYTHING BELOW A(I) IS LESS THAN OR EQUAL TO A(I)}
    IF A(I)>A(I+1)
      THEN T=A(I)
        A(I)=A(I+1)
        A(I+1)=T
      I=I+1
  {I=TOP}
  TOP=TOP-1
{TOP=1}
```

We know that the first invariant is true on entry because TOP is the highest index and there is nothing above it. The second invariant is also vacuously true on entry because I is 1 and there are no elements with smaller index. After execution of the inner loop, I has increased by one, but if the value of A(I) is greater than the value of A(I+1), then the larger value was moved up so that the invariant is still true. At termination of the inner loop, I equals TOP and in light of the invariant, everything below A(I) and thus A(TOP) has value less than A(TOP). Since everything above A(TOP) was in order and of value greater than A(TOP) by the outer loop invariant, we know that the next largest value is now in A(TOP). Therefore, we can see that after TOP is reduced by one, the outer loop invariant continues to be true. Upon termination of the outer loop, TOP is equal to one, and this, in light of the outer loop invariant, implies that the array is in ascending order.

The reader should realize that it is possible to present this reasoning in a more formal, rigorous manner. The outer loop invariant in the above example could have been written as:

```
[( I>TOP AND J>TOP AND I<J => A(I)<=A(J) ( AND
   ( I<=TOP AND J>TOP => A(I)<A(J) ]
```

While such formality may improve clarity and facilitate formal proofs, what is important is the style and strategy of the reasoning involved, and a programmer should not forego reasoning logically about the correctness of a program because of unfamiliarty or impatience with the formal expressions.

Exercises

1) Describe the functions of the following program fragments as transformations of the state of the computation.

a)
```
D = A-B
A = A/D
B = B/D
```

b)
```
IF (B-A)>(C-B)
    THEN N=C-B
      D=B-A
    ELSE N=B-A
      D=C-B
```

c) IF A>B
    THEN IF C>B
        THEN R=(C-B)/(B-A)
        ELSE R=(B-C)/(B-A)
    ELSE R=1

2) Determine whether or not the assertion within brackets is a loop invariant for the repetition, given that M>0.

```
A=N
B=M
WHILE B>0
  {A+B=N+M}
  A=A+1
  B=B-1
```

3) Find a loop invariant for the following, given that M>0.

```
A=N
B=M
WHILE B>0
  A=A-1
  B=B-1
```

4) Prove that the following computes C = X * Y .

```
C=0
A=X
B=Y
WHILE B>0
  WHILE (B/2)=INT(B/2)
    A=A+A
    B=B/2
  C=C+A
  B=B-1
```

1a)

statement	A	B	D
D = A-B	A0	B0	A0-B0
A = A/D	A1/D1	B1	D1
B = B/D	A2	B2/D2	D2

Then

A3 = A2	B3 = B2/D2	D3 = D2
= A1/D1	= B1/D1	= D1
= A0/(A0-B0)	= B0/(A0-B0)	= A0-B0

Then composite function is:

(A1,B1,D1) <- (A0/(A0-B0),B0/(A0-B0),A0-B0)

3) At the beginning of the loop, A=N and B=M. Therefore, A+B=N+M. After the execution of the body of the loop, A1=A0+1 and B1=B0-1. Therefore, A1+B1=(A0+1)+(B0-1)=(A0+B0)+(1-1)=A0+B0=M+N.

Thus, the proposed loop invariant holds at the beginning and after each execution.

4) Consider the loop invariant A+B=X*Y and make an argument similar to that given for problem 3.

References

For Further reading, refer to:

Borgerson, Mark J., *A BASIC Programmer's Guide to PASCAL*, John Wiley & Sons, New York 1982

Brown, P.J., *PASCAL from BASIC*, Addison Wesley, Menlo Park, CA 1982

Moulton, Peter, *Foundations of Programming Through BASIC*, John Wiley & Sons, New York, NY 1980

Welsh, Jim and Elder, John, *Introduction to PASCAL, 2nd Ed.* Prentice Hall, Englewood Cliffs, NJ 1982

Improving

in BASIC

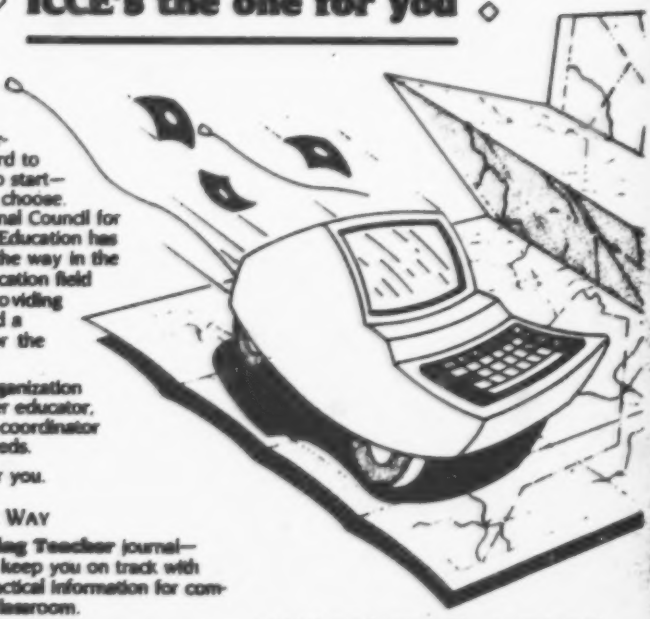
Sharon Burrowes
Ted Burrowes

ICCE Publications

$6.00

Take Off With Us

◇ ICCE's the one for you ◇



With today's profusion of computer information, it's hard to know where to start—which road to choose. The International Council for Computers in Education has been guiding the way in the computer education field since 1979, providing leadership and a ground plan for the future.

It's the one organization every computer educator, administrator, coordinator or librarian needs.

It's the one for you.

GUIDING THE WAY

The Computing Teacher journal—guaranteed to keep you on track with up-to-date, practical information for computers in the classroom.

Special Interest Groups—share information to help your special interest area grow. SIGs include computer coordinators, teacher educators, administrators and special educators, and are planned for advanced placement in computer science, community colleges and videodisc users. The quarterly **SIG Bulletin** serves as a forum for SIG information.

Booklets and Monographs point to additional information on specific topics. **ICCE Packets** provide you with teacher training materials. Members receive a 10% discount on all three.

ICCE Committees address a variety of ethical and practical issues important to you as a computer-using educator.

ICCE participates in computer education conferences throughout the world, supporting our state- and region-wide member organizations.

ICCE — Join the Core for You

ISBN 0-924667-28-1

BEST COPY AVAILABLE